ALS
ASCII Learning System

アスキー・ラーニングシステム　②実習コース

# 実習 C言語

三田典玄 著

アスキー出版局

アスキー・ラーニングシステム　②実習コース

# 実習
# C言語

三田典玄 著

アスキー出版局

---

■本文イラスト　皆川 幸輝

# はじめに

本書は「入門C言語」の続編として書かれていますが，ある程度C言語に関する知識がある方は，本書のみを読まれてもC言語についての知識が修得できるように構成してあります．

また，本書ではC言語の特徴である「構造化」という考え方とその考え方をデータ構造に反映させた「構造体／共用体」の説明，そして実際にプログラムを記述していく上で欠かせないさまざまな知識を具体的な例題とともに体系化しました．

本書を書く上でとくに重点を置いたのは以下のことです．

1．知りたいことがすぐ引けるようにリファレンス性を重視した．
2．図版を多用することで視覚的な理解ができるように心がけた．
3．多くの処理系で移植が可能なように，関数とインクルードファイルの対応表を掲載した．

最近では「C言語を従来のアセンブラの代わりに使う」という利用法も一般的になりつつあり，C言語はより高いソフトウェアの生産性に貢献しています．アセンブラを使用すると高速な演算処理が可能ですが，プログラムの記述が複雑になりがちです．逆に多くの高級言語では記述性は高まりますが処理速度に不満が残ることがあります．しかしC言語のようなプリミティブな高級言語を処理系として採用することによって，記述性のよさと高速な演算処理という相反する要求を少なからず満たすことができるのです．

C言語は最初，UNIXを記述するために書かれた言語ですから，オペレーティング・システムの記述に最適な言語構成となっています．しかし，最近では前述のようなC言語の特徴からアプリケーションプログラムの開発にもごく普通に使われるようになってきました．その影響で最新のC言語コンパイラでは標準ライブラリに非常に多くの関数が含まれています．また，このようにアプリケーションを書く上で便利な関数が増えたことで，C言語プログラマの数は年々増加しています．C言語は現在最も多く使われているパソコン上のコンパイラ言語です．

本書を活用することによって，より深いプログラミングへの理解とよりよいソフトウェアの開発がなされることを願ってやみません．

1986年9月28日　　三田典玄（asc11604／dwq@uupun.juice）

# C言語ラーニングシステム　全3巻の構成

C言語のアスキー・ラーニングシステムは，入門C言語，実習C言語，応用C言語の全3巻で構成されます．

従来のC言語の書籍は，主にコンピュータをよく知っている人たちを対象に書かれており，数多くのことがらが1冊の本につまっていました．このため基本的なコンピュータの知識を持っていない初心者が学習するにはかなり無理があったように思います．また逆に，コンピュータを使いこなしている上級者にとっては，細かい説明やくわしい内部構造など，ほんとうに知りたいことがいまひとつ載っていないことへの不満が少なからずあったことでしょう．

今回のシリーズでは，初心者から無理なく学習でき，また上級者にも満足できるようにとの主旨から全体を3巻にわけて構成しました．以下に各巻の概要を紹介しておきます．

**入門C言語**：本巻では，C言語で簡単なプログラムが書けるようになることをテーマとしています．また，C言語を学ぶ上で不可欠なコンピュータの基本構造についても解説し，この先より大きなプログラムが書けるようになるための基礎的な力をつけることを主眼に置きました．

**実習C言語**：入門編で取り上げなかった「構造体」，「共用体」などの学習と，C言語についてのすべての項目がわかるように体系的な整理を行います．また，各項目ごとに数多くの例題を挙げ，各自でプログラムを組んでいく際にマシンのそばに置いていつでも参照できるように構成します．

**応用C言語（発売予定）**：C言語を本格的に使いこなすために，さらに高度なプログラミングと開発環境について学習します．とくに，アセンブラの代わりとしてC言語を使う場合のプログラムのROM化の手法，高速化の手法，他言語とのリンク，バグの回避の方法など実践的なテクニックに加えて，オリジナル・ライブラリの作成，日本語処理のノウハウなども紹介します．

本シリーズを読むに当たっては，実際にコンピュータを目の前に置いて動かすことを前提としていますが，書籍を読むだけでも体験的な学習ができるように配慮しています．

# 目　次

# 掲載プログラム一覧

# 第1章
# C言語による
# システム設計

C 言語のプログラミングでは，COBOL や FORTRAN などの大型機の言語に比べて詳細なシステム設計をしないことが多いようです．とくにパソコン上の場合は，「ちょっと組んでみる」という程度のアプリケーションが多いからでしょう．しかし，実際に大きなプロジェクトでC 言語を使ってみると，数々の細かい不都合やライブラリなどのバグに悩まされることがあります．この場合，システムの設計がしっかりしていれば間違いを発見するのは容易です．また，そうでなければ「自分の間違いか，それとも処理系の間違いか」で悩まされることになります．つまり，「しっかりしたシステムは，（自分のバグであっても他人のバグであっても）その発見を早くする」というわけです．

いずれにしろシステム設計は，おろそかにしてはいけません．プログラムの生産性は，ひとえに「デバッグの効率」にかかっているといっても過言ではないのですから．

この章では，実際に仕事でプログラムを組む場合に，プログラマや SE がどのようにプログラムの設計をしているのかということを中心に解説します．このような話は，アマチュアでプログラムを組んでいる方々にもその技術向上の一助になるのではないかと思います．まずは気軽に読んでみてください．

# 1.1　C言語の特徴と開発環境

システム設計というと大規模なものを想定しがちですが，そればかりではありません．小さなコマンドを1つ作る上でもその考え方は非常に重要です．本書では，開発言語としてC言語を使っていきますので，システム設計について解説する前にC言語の特徴とその開発環境について再確認しておきましょう．

## ■C言語の特徴

プログラミング言語の世界でC言語の占める位置というのを図にすると以下のようになります(図1-1)．

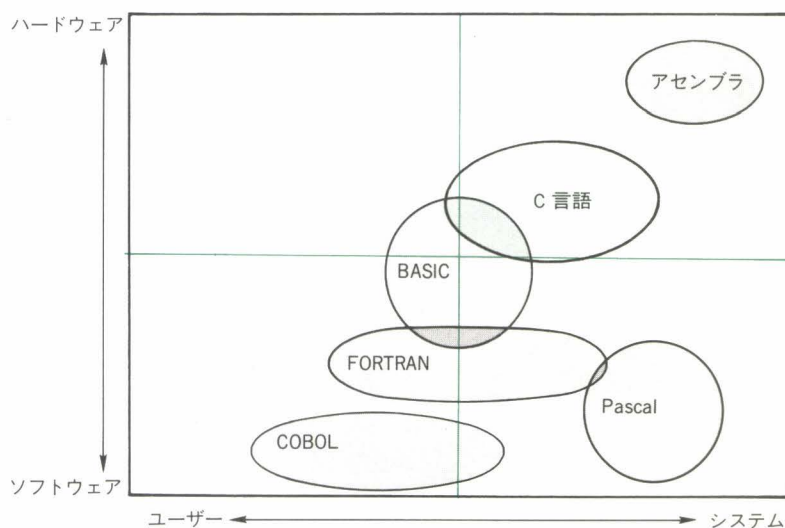



図1-1　C言語と他のプログラミング言語の関係

C言語は，ハードウェア寄りの構造化されたアセンブラのような言語です．それに対して昔からあるFORTRANやCOBOLといった言語はソフトウェアからハードウェアが見えない(見えなくてもよい)言語です．またPascalは，構造化されていてもハードウェアにはやはり直接のアクセスは不可能でした．つまりC言語とその系統にある言語(BCPLからの流れ)は，従来からのFORTRANやCOBOLなどの流れとは一線を画すものなのです．

近年コンピュータは，単なる電子計算機としての意義だけでなく，周辺機器を制御する「コントローラ」としての役割も大きくなっています．ところが，これまではコンピュータに付いたハードウェアを直接制御するために，コンピュータにじかに話しかける**機械語**が必要でした．この機械語は実にやっかいな言葉で，「覚えることや制限が多い」，「作ったプログラムが読みにくい」，「CPUによって言葉が違う」といった欠点があります．つまり，直接コンピュータが制御できるという特徴以外はあまり取柄がありません．

そこで，「どんなCPUでも同じように使え，ハードウェアのきめ細かい制御ができる」言語として（つまりパソコン用の言語として）白羽の矢が立てられたのが**C言語**というわけです．

さて，こういった経過で世の中に出てきたC言語ですが，プログラムを組む側から見ると以下のような特徴を持っています．

1．ハードウェアの直接アクセスがアセンブラのように可能
2．プログラムを組むに当たって覚える事柄が少ない
3．見やすいプログラムが容易に書ける
4．必要ならばアセンブラとのリンクも可能

すなわち，「汎用構造化アセンブラ」という位置付けができます．また逆に欠点としては，

1．いわゆる「ノイマン型コンピュータ」に関する一般的知識が必要†
　つまり，一度はアセンブラを組んだことがないと，C言語は容易に扱えない
2．行に捕らわれないプログラムが書けるため，下手をするとぐちゃぐちゃなプログラムになってしまう．あくまでも「読みやすいプログラムが書ける」のであって，その努力をしないと通常のアセンブラ以上に読みにくいプログラムになる

という点が挙げられます．

## ■C言語の開発環境

C言語は汎用のプログラミング言語ですから，あらゆるコンピュータのあらゆる種類のプログラムが組めます．そこでまず，「どんなコンピュータの上で」，「どんなプログラムを組むのか」，そして「どんなコンピュータ上で走らせるのか」ということを明確にしないといけません．また，UNIX，MS-DOS，CP/Mなどのソフトウェア環境が何であるかも重要な要素です．

---

† 前巻「入門C言語」では，C言語でプログラムを組むに当たって最低限必要なコンピュータの一般的知識について解説している．

たとえば,「16ビットパソコンを使って, Z-80CPUを使った制御専用のコンピュータ上で動くプログラムを書く」のか,「メインフレーム上で動くプログラムをMSXパソコンで組む」のか(冗談のようですが可能です), ということをしっかり把握しておく必要があります.

しかし, 多くのユーザーにとっては,「プログラムを組むのと同一のコンピュータ上で動くプログラムを書く」のが普通であり, またその多くはパソコン・ユーザーであると思いますので, 本書でもこの組み合わせを標準にしていきます.

**ハードウェア**

まずハードウェアは, **図1-2**のような組み合わせが必要です.

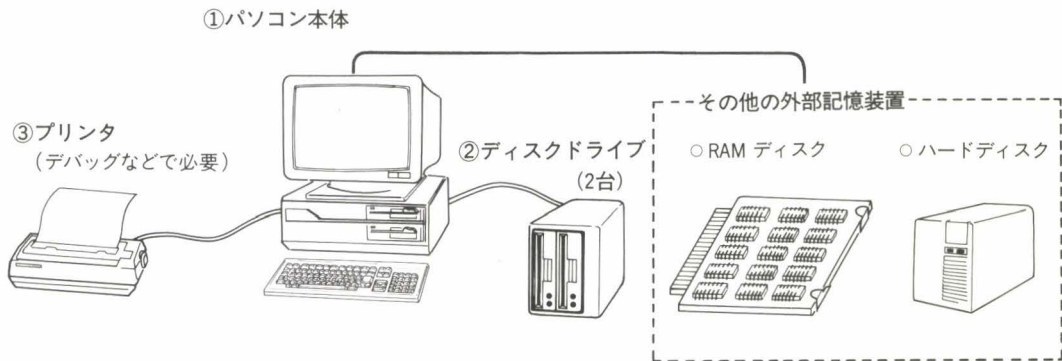



**図1-2　必要なハードウェア**

外部記憶装置は, **RAMディスク**や**ハードディスク**もあるといっそう使いやすい環境になります. とくに最近のコンパイラは, いくつものモジュールを次々に呼び出して実行するものが多く, またライブラリなどコンパイルやリンクのためにアクセスするファイルは, 数, 大きさとも非常に大きくなっています. このようなディスクアクセスを多く行うファイルは, これらの高速アクセス可能なデバイスに入れておくとコンパイルやリンクの時間が短縮されます.

たとえば, RAMディスクの容量が少ない場合でも, コンパイラのはき出すテンポラリファイルを作るドライブとしてRAMディスクを指定しておけばよいでしょう. また, ハードディスクがあれば, ライブラリなども含めて必要なファイルをすべてそこにほうりこんでおけるので, プログラムによっては, フロッピーディスクに比べておよそ5倍以上のコンパイル速度の改善が計れます. 今まで, コンパイルの遅さにいらいらしていた人はぜひ試してみるとよいでしょう. また, 仕事でC言語を勉強される方には, ハードディスクの使用をお勧めします.

## ソフトウェア

次にソフトウェアを揃えましょう．C言語の種類にもよりますが，たとえば**図1-3**のようなソフトウェアが必要です．

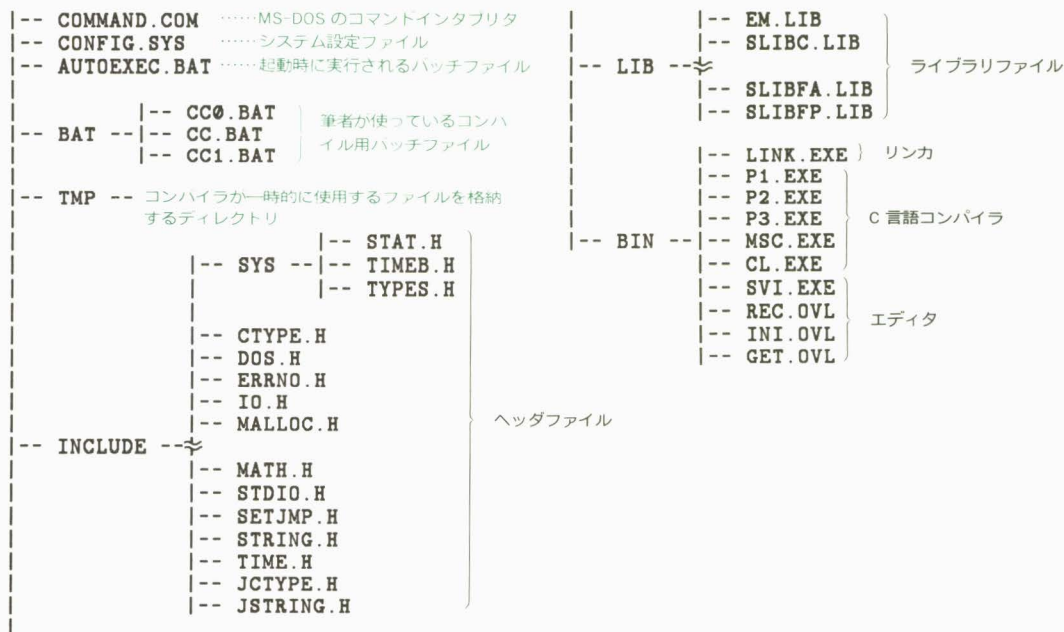



**図1-3　必要なソフトウェア**(Microsoft C Compiler **の場合**)

ソフトウェアでは，どのメーカーのC言語を使用するかという問題があります．これは，何を開発するのかなどによって異なりますので，ここでは追求しません．本書のサンプル・プログラムは，すべて標準的なC言語で記述しており，多くの処理系でそのまま動かすことができます．ただし，特定の処理系に依存してしまう部分は，各処理系の関数の対応表を巻末のAPPENDIXに掲載していますので，それを参考に移植してください．

さらに，MS-DOSの「CONFIG.SYS」や「AUTOEXEC.BAT」のようなシステム設定[†]に必要なファイルは，システムの使いやすさに大きく影響してきますので注意してください．

† 前巻「入門C言語」では，代表的な処理系とその環境の設定方法を解説している．

# 1.2　プログラム作成の実際

さて，それでは実際にコマンドを１つ作成していきながら，システム設計の基礎とＣ言語プログラミングの考え方などを学んでいくことにしましょう．

## ■ システム設計の概要

ちょっと大きなプログラムを組もうと考えると，かならずシステム設計なる段階を経なければなりません．プログラムを組む前にその目的や内容，実現可能性などの検討と設計にかかる大まかな時間を予想し，プログラムを組む時点での能率の向上を計ります．たとえアマチュアであって，そのプログラムを組む時間や工数等の制限がない場合でも，プログラムを組むという同じ行為を行うのですから(できあがったシステムの有用性はともかく)，これらの作業はやっておくにこしたことはありません．

システム設計と一口にいっても，さまざまな方法や書式があります．とにかく必要なことは，おおまかに分けて，

1．プログラムが何に使われ，何にとって有用なのかを明らかにする
2．使用できる環境を明らかにする
3．実現の方法を明らかにする

ということになります．また，時間的な流れを追ってみると，プログラム完成に至る道筋は一般的に次ページの**図1-4**のような手順をとります(これはプロの場合です)．

プログラムが大きく，かなりの人数でそのプログラムを仕上げなければならない場合，このような**仕様書**の存在はかなりのウエイトを占めます．しかし，実際にこの本をお読みの方々のなかには，いちいちこんなことはやっていられないという人も多いでしょう．実際，プログラムを１人でコツコツと作る場合は，①から③の過程は省略されるのが普通です．しかし，①から③については，自分でいつ見てもわかるメモ程度のものは残しておくことをお勧めします．これらの「メモ」の存在は，デバッグの段階で「いったい自分は何をしようとしていたのか」がわからなくなった場合(よくあることです)に絶大な威力を発揮します．

システム設計は以上のような流れをとりますが，これらの過程で本当にＣ言語は必要なのかを検討していく必要があるのはいうまでもありません．本書ではその例題に「Ｃ言語での開発が最適なもの」をあらかじめ選択してありますが，実際の場面ではそうとばかりは限りません．Ｃ言語の特性を承知した上での「賢明な選択」が，コンピュータのプログラムを成功させる最初の鍵の１つです．

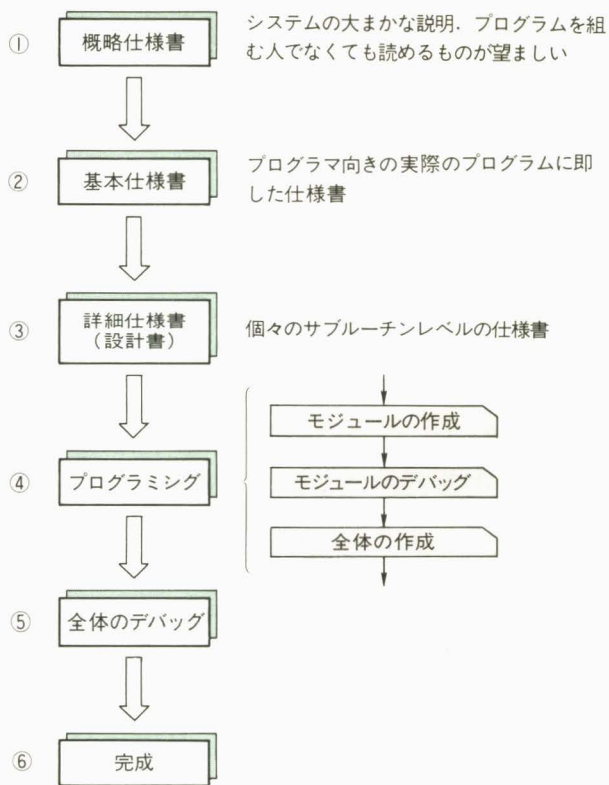



**図1-4　プログラム完成までの手順**

## ■ プログラムの仕様の決定

まず，「何を目的としたプログラムを作るのか」を決めるのが最初です．ここでは「指定された複数のファイルのなかの最初の数行を見るプログラム」を作ってみましょう．最近流行りの「イントロ機能付きラジカセ」のコンピュータ版というわけです．

自分でいろいろなプログラムやファイルを作っていると，いつのまにか「あれ？ このファイルは何だったっけ」という事態によく遭遇します．また，その管理のためにあるファイル名もいいかげんになりがちです．そんな時，「ファイルを最初からちょっとずつ見ていく」プログラムがあるとたいへん便利です．

**表1-1** にその仕様を示します．

| プログラム名 | chotto　（ちょっと表示するという意味） |
| --- | --- |
| プログラムの機能 | 入力されたファイルの最初の数行だけを表示する．表示されるファイル名は，一度に複数の入力を可能とする |
| 開発環境 | C言語でプログラミングが可能なオペレーティング・システム上 |
| 起動方法と使用方法 | A＞chotto ↵ ……プログラムの起動<br>FILE-NAME ＝ test 1 ↵<br>FILE-NAME ＝ test 2 ↵　} ファイル名の指定<br>⁝<br>FILE-NAME ＝ ↵…………ファイル名の入力終了<br>LINES ?　＝ 10 ↵………行数の指定 |
| 使用上の注意 | 2バイトコード(漢字など)が含まれたファイルはサポートしない<br>テキストファイルでない場合は，アスキーキャラクタに置き換えて表示する |

**表 1-1　chotto プログラムの仕様**

本来ならばファイル名を入力しなくても，あるディレクトリの下の読めるファイルはすべて表示する同様のコマンドとしたかったのですが，ここでは，

1. ディレクトリの構造やその解析方法がオペレーティング・システムによって違う
2. コマンドの引数(コマンドに続く文字列)の扱いをまだ学習していない
3. システム設計の本質を理解してもらうためにプログラム自身はできるだけシンプルにしたい

という理由で，ここでは見送りました(コマンドの引数の扱い方は第5章に出てきますので，あとで各自で改良してください)．

## ■ プログラムのアルゴリズムとフローチャート

プログラムのなかでいったい何が行われているのかを書き表すのには，やはりフローチャートが多く使用されます．最近ではNSチャートなどの構造化プログラミングに適したアルゴリズム記述用の各種のチャートが考えられ発表されていますが(そして一部では使われていますが)，なかなか浸透していないのが実状です．そこで，本書でもおそらく多くの人が使い慣れているはずのフローチャートを使っておきます．

次ページの**図 1-5** に，「chotto」プログラムの関数別のフローチャートを示します．chotto プログラムは，「main()」，「input_file()」(ファイル名の入力)，「input_line()」(行数の入力)，「disp_file()」(複数のファイルの内容を表示)，「disp_1file()」(ファイルの内容を表示)，「cat_file()」(実際の表示の処理)の6つの関数から構成されています．

図1-5　chotto プログラムのフローチャート(1)

〈cat_file 関数〉

cat_file（ ）

i = 0

i > MAX

YES

指定行数を越えたか？

disp_1file 関数へ

NO

fgets

1 行読み込み

j = 0

j > MAX

YES

1 行の文字数を越えたか？

NO

0？

YES

j 文字目は 0(NULL)か？

NO

DISP？

NO

表示可能な文字か？

YES

putchar

1 文字表示

j ++

i ++

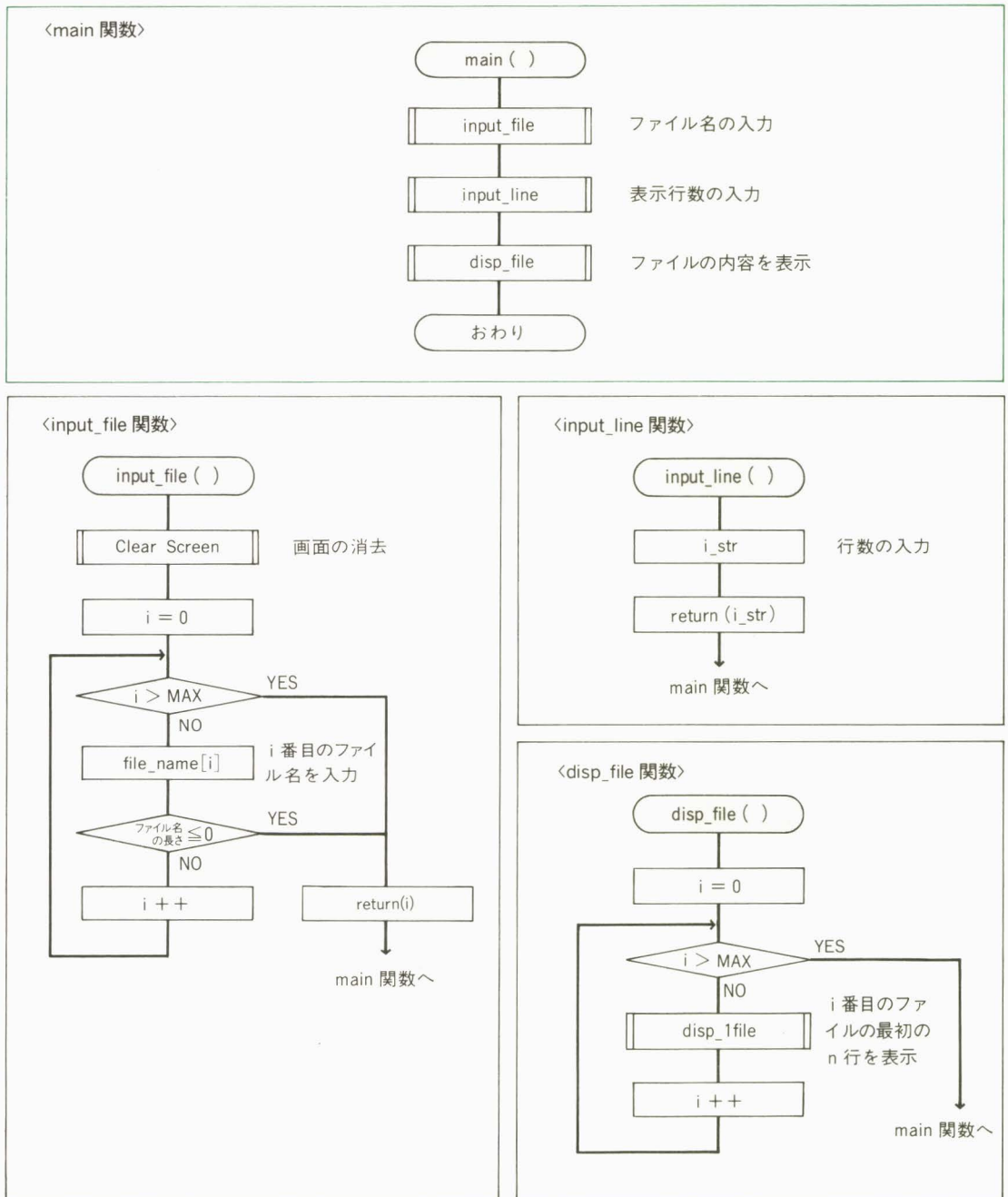
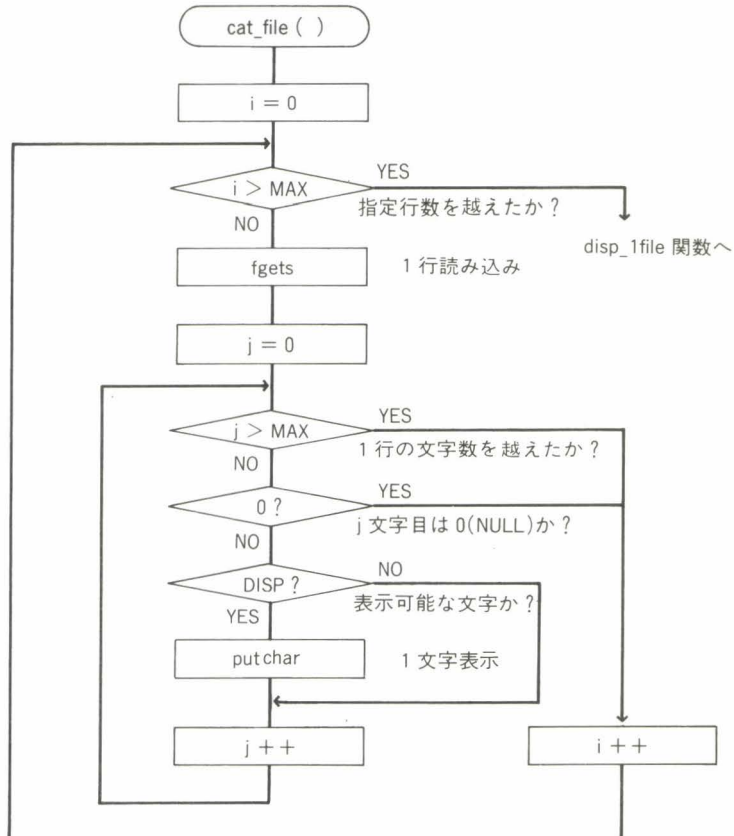



図 1-5　chotto プログラムのフローチャート(2)

フローチャートを見ると，メインルーチン(main関数)から必要な関数が次々に呼び出されていることがわかります．とくに，実際の画面表示を行うdisp_file関数は，そのなかでdisp_1file関数を呼び，さらにそのなかでcat_file関数を呼ぶというように実行する単位が小さくなっていきます．**図1-6**にこれらの関数の呼び出し関係をまとめます．

また，最後のcat_file関数は，万一指定されたファイルがテキストファイルでなかった場合のために，ディスプレイに表示できない文字を表示しないようにする処理を含んでいますのでちょっと複雑になっています．

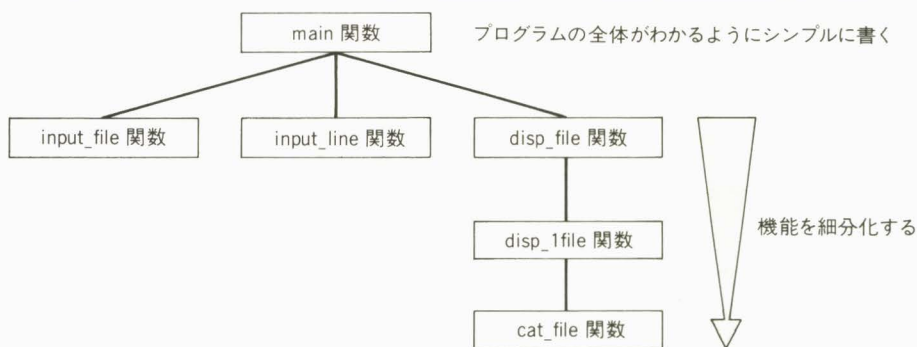



**図1-6　関数の呼び出し関係**

## ■ プログラムリスト

先のフローチャートからできるプログラムを**リスト1-1**に示します．

個々の関数の仕様については，とくに難しいものはないので省略します．また，関数の内容についても説明しませんので，各自で解析してみてください．

```
 1: /*
 2:
 3:   Chotto.c    ------ multifile cat. procedures.
 4:
 5:   Designed & created by N.Mita 1986/05/28
 6:   Copyright CoreDump Co,.Ltd.
 7:
 8: */
 9:
10:
11: #include     <stdio.h> ··················stdio.h ファイルの取り込み
12: #include     <ctype.h> ··················ctype.h ファイルの取り込み
```

```
13:
14: #define    CLS    cls() ………………画面を消去する関数を呼ぶマクロ定義
15:
16: #define    MAXFILES     128 ……このプログラムで扱えるファイル数の最大値
17: #define    MAXLINES     256 ……このプログラムで扱える1行の文字数の最大値
18:
19: static char    file_name[MAXFILES][40]; ………ファイル名を入れておくバッファ
20: FILE           *fp;                      ………ファイル構造体へのポインタ
21:
22:
23: cls() ………画面を消去する関数(どの処理系でも対応できるように, 25 行空行を送るだけにしてある)
24:     {
25:     int    i;
26:     for(i = 0 ; i < 25 ; ++i) printf("¥n");
27:     }
28:
29:
30: int     open_file(fname) ………ファイルをオープンする関数
31:         char    fname[];
32:         {
33:         if(NULL == (fp = fopen(fname, "r"))) return(0);
34:         else                                 return(1);
35:         }
36:
37:
38: int     close_file() ………ファイルをクローズする関数
39:         {
40:         fclose(fp);
41:         }
42:
43:
44: int     cat_file(line) ………ファイルの内容を指定行数だけ表示する関数
45:         int     line;                    /* number of display lines */
46:         {
47:         int     i, j;
48:         char    line_buff[MAXLINES]; /* display line-buffer      */
49:
50:         for(i = 0 ; i < line ; ++i) ………1行を処理するループ
51:             {
52:             if(NULL == fgets(line_buff, MAXLINES, fp)) ………1行を読み込み,エラーが起きたかどうか判断する
53:                 {
54:                 if(feof(fp)) break; ………ファイルの終わりならば for ループを抜ける
55:                 if(ferror(fp))
56:                     {                                                 エラーであれば
57:                     printf(">>>>>>> Read-ERROR on FILE <<<<<<<¥n");   処理を中断して
58:                     break;                                            終了
59:                     }
60:                 }
61:
62:             for(j = 0 ; j < MAXLINES ; ++j) ………ファイルから読み込んだ行を表示する
63:                 {
64:                 if(NULL == line_buff[j]) break;                     読めない文字が
65:                 if((isspace(line_buff[j])) || (' ' <= line_buff[j]))  あった場合は,
66:                     putchar(line_buff[j]);                          「.」(ピリオド)
67:                 else                                                を表示する
```

```
 68:                     putchar('.');
 69:                 }
 70:             }
 71:     }
 72:
 73:
 74: int     disp_1file(fname, n) ………指定されたファイルを表示する関数
 75:         char    fname[];     /* file-names for display  */
 76:         int     n;           /* number of display-lines */
 77:         {
 78:         if(!open_file(fname))
 79:             {
 80:             printf("******* Cannot open file : %20s *******¥n", fname);
 81:             return(0);
 82:             }
 83:         else
 84:             {
 85:             printf("------- FILE : %20s -------¥n", fname);
 86:             cat_file(n);
 87:             printf("¥n¥n");
 88:             }
 89:
 90:         close_file();
 91:         return(1);
 92:         }
 93:
 94:
 95: int     disp_file(m, n) ………指定された複数ファイルを表示する関数
 96:         int     m;          /* Number of files */
 97:         int     n;          /* Number of lines */
 98:         {
 99:         int     i;
100:         char    dummy[40];
101:
102:         for(i = 0 ; i < m ; ++i)
103:             {
104:             disp_1file(&file_name[i][0], n);
105:             puts("Press return ..........");
106:             gets(dummy);
107:             }
108:         }
109:
110:
111: int     input_file() ………ファイル名の入力関数
112:         {
113:         int     i;
114:
115:         CLS;
116:
117:         for(i = 0 ; i < MAXFILES ; ++i)
118:             {
119:             printf("%5d : FILE-NAME = ", i+1);
120:             gets(&file_name[i][0]);
121:
```

```
122:            if(0 >= strlen(&file_name[i][0])) break;
123:            }
124:
125:        return(i);
126:        }
127:
128:
129: int    input_line() ………表示行数の入力関数
130:        {
131:        char    i_str[40];
132:
133:        printf("       : LINES  ?  = ");
134:        gets(i_str);
135:
136:        return(atoi(i_str));
137:        }
138:
139:
140: main() ………メインの関数. できるだけシンプルになっていることが望ましい
141:     {
142:     int    i, j;
143:
144:     if(0 >= (i = input_file())) ………ファイル名の入力
145:         {
146:         printf("¥7******* No files *******¥n");
147:         exit(0);
148:         }
149:     else
150:         {
151:         if(0 >= (j = input_line())) ………表示行数の入力
152:             {
153:             printf("¥7******* No lines *******¥n");
154:             exit(0);
155:             }
156:         disp_file(i,j); ………実際の処理
157:         }
158:
159:     printf("------- Complete.-------¥n"); ………作業終了の表示
160:     }
```

リスト 1-1　chotto プログラムのリスト

# 1.3　プログラミングのスタイルと考え方

前節で作ったプログラムを基に，C言語プログラミングのスタイルや書き方を分析してみましょう．読者の方の考え方とは異なる点があるかもしれませんが，1つの指針として読んでみてください．

## ■ プログラミングのスタイル

C言語は識別子や予約語の区切りとして，スペース，タブ，改行などの空白文字を任意の個数入れることができます．つまりBASICなどの他の言語と異なり，「行」に捕らわれない自由な書き方が可能です．このことを**フリーフォーマット**と言います(くわしくは，「4.1　C言語の構成要素」を参照のこと)．

たとえば，同じ内容を**表1-2**のように書き表すことができます．

注：C言語の制作者であるKernighanとRitchieの著書「プログラミング言語C」(共立出版)のこと

**表1-2　プログラミング・スタイルの例**

さて，それぞれのプログラムは，書いた本人が書きやすく読みやすいつもりでも他人には読みにくかったりすることもあります．このあたりは，プログラマそれぞれの個性がにじみ出るところで，人間的な部分でもあります．しかし，注意深くこれらのリストを見てみると，以下のような分析も可能です．

①と③は，プログラムの構造に重点を置いて記述してある．リストを遠くから眺めたとき，その構造がはっきりするように書かれている．このうち①は構造の区切り「{}」を，③は実行文の位置を重要視している．

②は，変数名や実行文の意味を中心に書かれている．構造は文によって規定され，①に比べて「{}」にあまり重きを置いていないが，変数名，関数名などの意味を中心にその処理の内容を読むことができれば，わかりやすいリストといえる．

また，④は一文で何をやっているのかを記述しているので，その実行文が短ければ最もすっきりした形である．

いずれにしろ，このようなプログラミングのスタイルには規制はありません．自分がわかりやすいように記述すればよいのです．

本シリーズでは，プログラミングに際してC言語の構造を中心に考えていきますので，基本的に①の記述方法を取っています．

## ■ 関数の考え方

C言語では関数のなかの実行の単位をどんどん重ねていける(このことを**ネスティング**といいます)ので，1つの関数ですべてを記述することもできます．しかし，これでは長くて見にくいプログラムになってしまい，またデバッグも大変です．そこで，**リスト 1-1** に示したプログラムのようにどんどん小さな関数にその役割を分けていきます．実際にプログラムを組む場合，1つの関数の大きさはCRTの画面1つに入るくらいが理想です(**図 1-7**)．あまり長い実行単位や関数の使用は，かなりプログラミングに慣れた場合でもしないように心がけましょう．

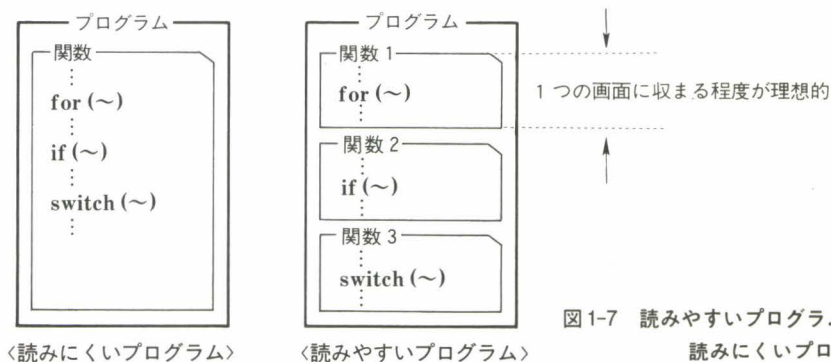



**図 1-7　読みやすいプログラムと読みにくいプログラム**

## ■ main 関数の位置　— 呼ぶ側と呼ばれる側の関数の位置関係 —

最近では，やっと main 関数の位置についてとやかくいわないようになってきましたが，一時は「main 関数はかならずプログラムの先頭になければならない」といわれていた時期がありました．これは K&R の「プログラミング言語C」という UNIX 上のC言語を解説した本(最初はこれしかなかった)で，そのような書き方がされ，奨励されていたので，「これがC言語の正しい書き方だ」ということになっていたためです．また，同じ本のなかで「C言語は**トップダウン**で書く言語だ」という説明があり，いっそうそのような書き方が広まったようです．

しかし，実際のコンパイラでは**図1-8**のように main 関数を上に書いておくと，int 型以外の返値(retrun value)を持った関数を使用する場合，その関数の宣言をあらかじめ使用する側の関数の内部でしなければならないという「めんどうな作業」が必要になります．そうでないコンパイラもありますが UNIX 準拠と銘打ったものはほとんどのもので必要です(第6章 関数を参照のこと)．

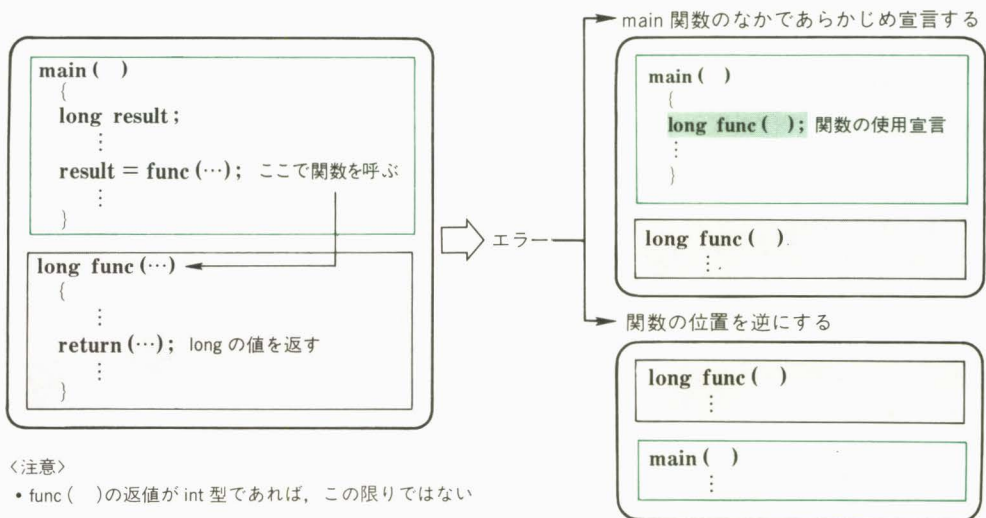



**図1-8　main()の位置**

研究者やアマチュアにとってはその目的の一部(あるいは全部)が「学習」にあるのですから，外部関数の使用宣言などは明確にし，できれば一点の曇りもないプログラミングを心がけるべきです．しかし，実際にC言語を使ってアプリケーションの開発等を行っている現場のプログラマにとっては，「時間の節約」も重要な課題の1つですから，　いちいち関数内での使用外部関数の宣言などをしていられない(1つの関数が 100 からの外部関数を使うこともある)場合もあります．

また，トップダウンかボトムアップかという問題は，仕様を決めるときに問題となるのであってプ

ログラムのコーディング・レベルでは関係ありません．

これらの問題は，前項の「プログラミングのスタイル」のところで述べたように，実はC言語を使う人の立場によってかなり違ってくるものです．要は自分の一番やりやすいスタイルを自分で見つけることです．その際，

1．たとえば1年後になって自分がリストを見てもわかるように書かれているか
2．小さなプログラムを書く場合でも，そのやり方が大きなプログラムで通用するように書かれているか

などをチェックポイントとするのがよいでしょう．

## ■ コメントについて

プログラムを組む場合，プログラムそれ自身のみではなくコメントを付けるのが普通です．これは他の言語でもまったく同じです．コメントについての守らなければならない原則はたった1つ，すなわち「コメントはできるだけ多く書く」[†]ということです．

またC言語の場合は，コメントの記号をどこにでも入れることができますから，かなり有効なデバッグの手段として使えます．たとえば，デバッグ中は動かしたくない関数や関数の一部などは「/*」と「*/」で囲っておけば，リストはそのままでも，コンパイラがコンパイルの対象からはずしますから余計なコンパイル時間の節約などにもなります．

## ■ プログラムの書き方

プログラムリスト（ソースリスト）は，プログラミングという作業の最終的な結果を記したものです．文章と同じですから，人によってきれいな文章とそうでない文章があり，読みやすいものもあればそうでないものもあるわけです．しかし，プログラムはそのセンテンスの1つ1つに意味があり，あいまいなセンテンスは1つもありません．

「とにかく動けばよい」というプログラムに限って，デバッグの時間を多く取ってしまいリストもきたないものです．また，きたないリストはあとで自分でも読む気がしないのが普通です．したがってデバッグの効率も落ちます（正直なところ，私自身も自分で書いていて耳が痛い話です）．

プログラムリストに関していえば，これは「他人の見るものであるから，できるだけきれいに」というのが原則です．たとえ他人が見ることがなくても，「来年の自分は今の自分ではない」のですから．

---

† 本書のプログラムでは読みやすさを重視しているので，多くの場合「/*　*/」を使ったコメントではなく解説として日本語でリスト中に示しています．

# 1.4　実行結果の確認とデバッグ

プログラムを打ち込み終ったら，以下の手順でデバッグと実行結果の確認をしてみましょう．デバッグなどは，エラーがでなければやらないという人も多いと思いますが，まずはここで示すとおりにやってみてください．

## ■ デバッグの方法

プログラムが書けたら，すぐに動くかどうか試してみたくなるのが人情ですね．とくにBASICに慣れた人はどうしてもすぐにコンパイルし，実行させせるのが習慣になっていないでしょうか？　このようなデバッグ方法は，C言語に関していえばコンパイルとリンクの時間ばかりかかってしまい，たいへん効率が悪くよくない方法です．

まずプログラムができあがったら，プリンタに出力し自分のプログラムを他人のそれと思ってデバッグしましょう．また，この段階でコメントに必要だと思われることが抜けていたら，どんどん追加しましょう．

次ページの**図1-9**にデッバグの手順を示します．この図でわかるように，プログラムのコーディング，コンパイル，リンク，実行という各段階で，異なった視点からのデバッグが必要になります．

## ■ 机上デバッグ

デバッグの最初は，かならず「紙の上で」行いましょう．とくにC言語では，数値のオーバーフロー(桁あふれ)など，BASICならば常識として行っている数々のチェックをコンパイラが行っていません．つまり，実行時に起こるエラーの管理をC言語はしていないのです．したがって，実行時に起こるはずのエラーの数々を念頭に置きながらプログラムを書くことが必要になります．「なんだ，面倒臭いんだな」と思われるかもしれませんが，実際にはこれらの「面倒臭さ」とC言語のよさはうらはらなのですから，これは避けて通れません．

デバッグとは一言でいえば，「自分自身の間違いを自分で発見し自分で直す」ということです．もちろん，人間というのは間違いがつきものですから，本当に「最初から最後まで」というわけにはいかなくて当然です．したがって，自分自身の間違いはできるだけ自分で見つけ，どうしても見つからないものについては，機械に頼るという方法が望ましいでしょう．またこのやり方は，きちんと完成したプログラムを作るための一番早い方法です．

とにかく「プログラムができた」といって，すぐに「コンパイル，実行」というのは愚の骨頂だということを忘れないでいてください．

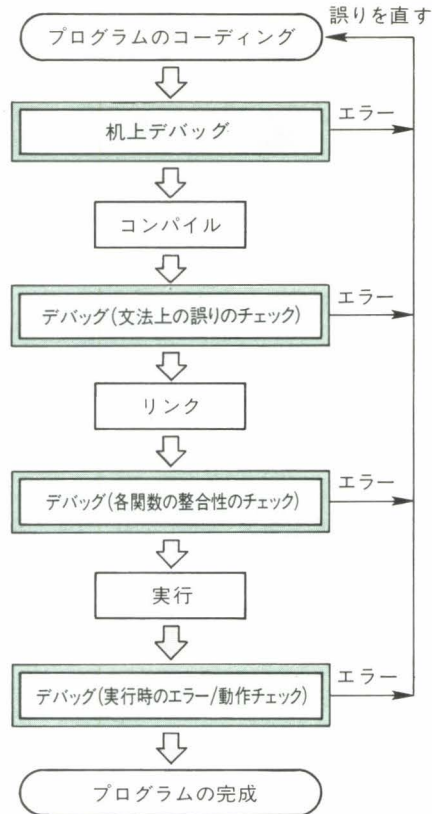



**図 1-9　デバッグの手順**

## ■ コンパイル時のエラー

さて，紙の上でのデバッグがもう完全だと思われるに至ったら，いよいよコンパイルを行います．このときエラーがあれば，コンパイラからエラーメッセージが出力されます．その場合には，そのエラーメッセージをよく読んで，対策をしなければなりません(メッセージの意味がわからなければ，辞書を引きましょう)．

コンパイル時のエラーは，文法的な誤りがほとんどですので，それほどデバッグに手間どることはないでしょう．とくに「{}」の対応や「,」(カンマ)，「；」(セミコロン)など構造の区切りとなる記号の抜けがないかどうかをチェックしてみてください．コンパイル時のデバッグ方法については前巻「入門C言語」でくわしく解説してあります．

## ■ 実行時のエラー

実行時のエラーは，

1．何がなんだかわからない結果が出てくる
2．システムが暴走する

という2つのパターンで出てくることがほとんどです．いずれもそのデバッグは非常に難しいことが多く，運悪くそのような結果に遭遇した場合は，覚悟したほうがよいでしょう．また，コンパイラ自身のバグなどもこのパターンで出てくることがあり，自分ではデバッグ不可能な状態に陥る場合もなきにしもあらずです．もちろん，だからといってバグが出て「これは自分では直せない」という事態にでくわした場合，すぐに「これはコンパイラのバグだよ」といって逃げるなどとはもってのほかです．こんな場合，筆者の経験から言うと「処理系の間違いはまずないと思ってよい．間違えているのはプログラムを書いた自分自身だ」ということがほとんどです．また「できない奴ほど，他人(コンパイラを作った人)のせいにしたがる」という経験則もあります．

プログラムを作る段階において，ちゃんと気をつけていれば(またマニュアルをきちんと読んでいれば)回避できるバグは多いものです．とくにアルゴリズムに関するバグ，オーバーフローに関するバグなどは，実行時に起こりやすいものですが，これらのバグは「すべて自分が至らなかったために起こったものだ」という認識は心にとめておいてください．

## ■ 実行結果

さて，これまでの手順でデバッグが終了すると，chotto プログラムは図1-10のように実行できます．

```
A>chotto↵
                        画面をクリア
    1 : FILE-NAME = chotto.c↵
    2 : FILE-NAME = b:buff.c↵
    3 : FILE-NAME = b:¥src¥tty.c↵  ………パス名を含んだ指定もできる
    4 : FILE-NAME = ↵  ……………リターンのみでファイル名の指定終了
      : LINES  ?  = 10↵  ……………表示する行数の指定
------- FILE :             chotto.c -------
/*

  Cyotto.c   ------ multifile cat. procedures.

  Designed & created by N.Mita 1986/05/28
  Copyright CoreDump Co,.Ltd.

*/
```

```
Press return .......... ↵

------- FILE :              b:buff.c -------
/*
        buff.c - a parts of les
                UNIX(LOCAL) like less command ultra super sub set
*/

#include <stdio.h>

#ifdef MSC
#include <errno.h>
#endif


Press return .......... ↵

------- FILE :          b:¥src¥tty.c -------
/*
        tty.c - a parts of les
                UNIX(LOCAL) like less command ultra super sub set
*/

#include <errno.h>
#include <signal.h>

#ifdef MSDOS
# include <fcntl.h>


Press return .......... ↵

------- Complete.-------

A>
```

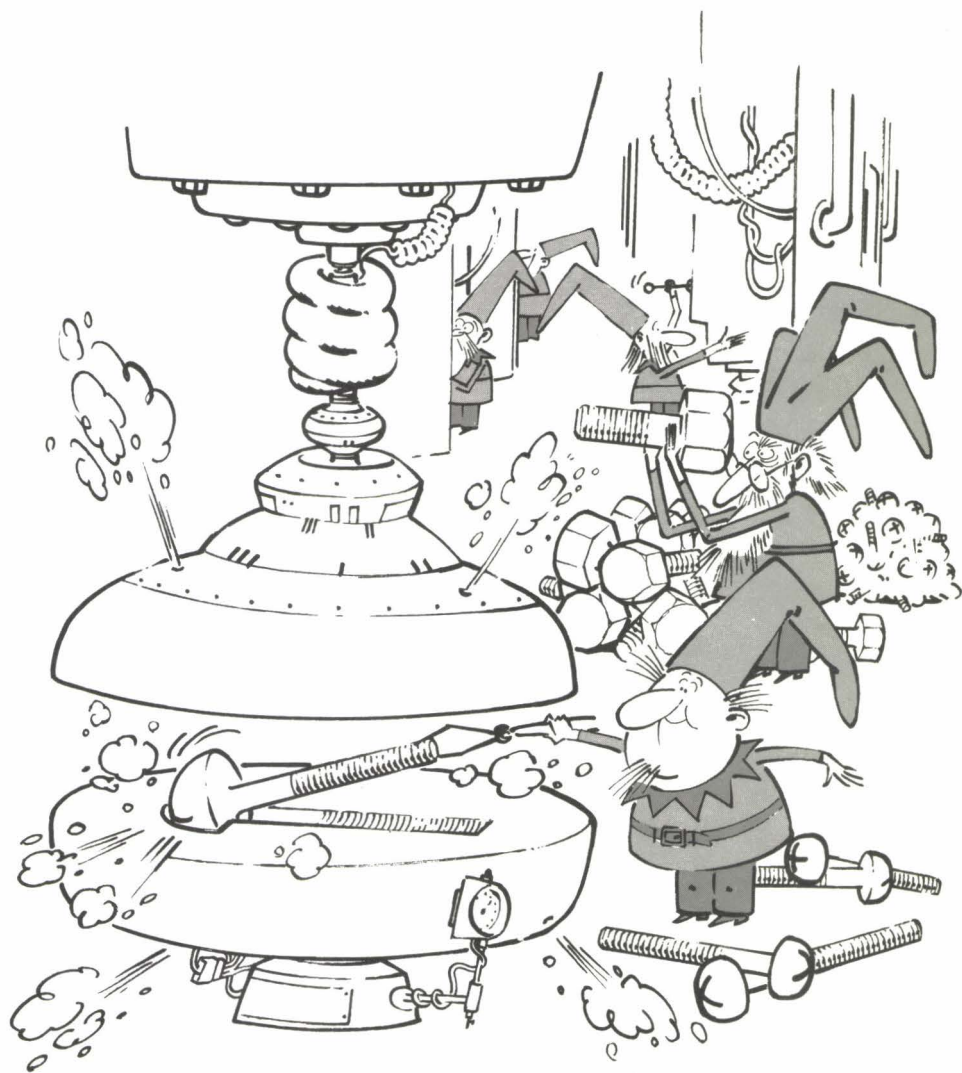
図 1-10　chotto プログラムの実行結果

## ■ システム設計のまとめ

システムの構築，設計，開発は，

1. 基本設計
2. プログラミング(コーディング)
3. デバッグ

がすべてです．プログラミングだけでなく，基本設計やデバッグに習熟することがプログラム開発の効率を上げ，プログラミングの楽しみを増やす道です．そのためにもより多くの「経験」を積むように努力してください．

# 第2章
# 演算子

C 言語には各種の演算子があり，その数は他のプログラミング言語に比べるとかなり多くなっています．これらの演算子は，演算の高速化を計ったり，きめ細かな制御をするために用意されたものが多く，C 言語の特徴の 1 つにもなっています．

またC 言語では，AND や OR などの論理演算を通常使われる算術演算と同様に扱います（論理演算のために使っている変数も，数値演算をする変数をそのまま使います）．つまり，論理演算と算術演算をごちゃごちゃに使うことができるので，それを使う側が意識して使い分けをしないと，わかりにくいプログラムができあがってしまいます．この点には，十分注意してください．

さて本章では，C 言語で使えるすべての演算子とその優先順位についてまとめていきます．とくに，アセンブラを知らないと理解しにくい「ビット演算子」とデータ型を変換する「キャスト演算子」については，少しくわしく取り上げておくことにしましょう．

# 2.1　C言語で使える演算子

C言語の演算子は，大きく分けて以下の3つに分類できます．

1．算術演算子
2．論理演算子
3．その他の演算子

これらの各演算子は，とくに使い方が難しいということはないので，以下ではその種類と使用法を表にしてまとめておくだけにします．

## ■ 算術演算子

算術演算子は通常の演算に使われる演算子で，〝加減乗除〟の演算子のほかにC言語に特有の便利な演算子があります．

### ― 二項演算子 ―

どのプログラミング言語にもある加減乗除のための演算子です．これらの組み合わせの優先順位は，すべて通常の演算と同じです．

**表2-1** に二項演算子の種類と使用法を示します．

| 演算子 | 意味 | 使用法 | |
|---|---|---|---|
| ＋ | 加算 | a＝b＋c； | bとcを加えた値をaに代入する |
| − | 減算 | a＝b−c； | bからcを引いた値をaに代入する |
| ＊ | 乗算 | a＝b＊c； | bとcを掛けた値をaに代入する |
| / | 除算 | a＝b／c； | bをcで割った値をaに代入する |
| % | 剰余算 | a＝b％c； | bをcで割った余りの値をaに代入する |

〈注意〉
- 整数（char, int, long, およびこれらのunsigned）の除算の結果に余りが生じた場合は切り捨てになる．
- 浮動小数点数での剰余算はエラーとなる．
- 原則として演算結果のオーバーフローの処理は行わない．

**表2-1　二項演算子**

**― 単項演算子 ―**

C言語の特徴の1つであるインクリメント演算子，デクリメント演算子があります．この演算子があるために，コンパイラはよりコンピュータに密着した，効率のよいオブジェクト・コードを作ることができます．また慣れてしまえば，簡潔な表記になりプログラムが読みやすくなります．

**表 2-2** に単項演算子の種類と使用法を示します．

| 演算子 | 意味 | 使用法 | 他の演算子による表記 |
|---|---|---|---|
| ＋＋ | インクリメント | a＋＋；または　＋＋a； | a＝a＋1； |
| －－ | デクリメント | a－－；または　－－a； | a＝a－1； |
| － | 符号の反転 | b＝－a； | ―― |

〈注意〉
• 単項演算子はこのほかに，「＊」(ポインタ)など演算子としてとくに意識されないものもある．

**表 2-2　単項演算子**

**― 代入演算子 ―**

代入演算子は，演算した結果を被演算数の1つに入れ直してくれます．この演算子には，あとで述べるビット演算子もありますが，ここでは算術演算と論理演算で共通に使われる代入演算子を**表 2-3** に示します．

| 演算子 | 意味 | 使用法 | 通常の算術演算子による表記 |
|---|---|---|---|
| ＝ | 右辺から左辺への代入 | a＝b； | ―― |
| ＋＝ | 左辺の変数と右辺の変数を加算し，左辺に代入 | a＋＝b； | a＝a＋b； |
| －＝ | 左辺の変数から右辺の変数を減算し，左辺に代入 | a－＝b； | a＝a－b； |
| ＊＝ | 左辺の変数と右辺の変数を乗算し，左辺に代入 | a＊＝b； | a＝a＊b； |
| /＝ | 左辺の変数を右辺の変数で除算し，左辺に代入 | a/＝b； | a＝a/b； |
| %＝ | 左辺の変数を右辺の変数で割った剰余を，左辺に代入 | a%＝b； | a＝a%b； |

〈注意〉
• ここでは算術演算，論理演算に使われるもののみを示した．

**表 2-3　代入演算子**

## ■ 論理演算子

論理演算子は，真あるいは偽という論理判断の結果を返します．これらはおもに制御文の条件として用いられます．

### ― 関係演算子 ―

2つの数の大小関係を評価する演算子です．演算子で表す関係が成立していれば(真ならば)値として1を，成立していなければ(偽ならば) 0 を持ちます．

**表 2-4** に関係演算子の種類と使用法を示します．

| 演算子 | 意味 | 使用法 | |
|---|---|---|---|
| > | より大 | a = (b > c); | cよりもbが大きければaに1を入れ，そうでなければaに0を入れる |
| < | より小 | a = (b < c); | cよりもbが小さければaに1を入れ，そうでなければaに0を入れる |
| >= | より大か等しい | a = (b >= c); | cよりもbが大きいか等しければaに1を入れ，そうでなければaに0を入れる |
| <= | より小か等しい | a = (b <= c); | cよりもbが小さいか等しければaに1を入れ，そうでなければaに0を入れる |
| == | 等しい | a = (b == c); | bとcが等しければaに1を入れ，そうでなければaに0を入れる |
| != | 等しくない | a = (b != c); | bとcが等しくなければaに1を入れ，そうでなければaに0を入れる |

**表 2-4　関係演算子**

### ― 論理演算子 ―

論理演算を行う演算子です．演算の結果は「真のとき1」，「偽のとき0」になります．

**表 2-5** に論理演算子の種類と使用法を示します．

| 演算子 | 意味 | 使用法 | |
|---|---|---|---|
| && | AND | a = b && c | bとcがともに真ならばaは真(1)．それ以外は偽(0) |
| ¦¦ | OR | a = b ¦¦ c | bまたはcが真ならばaは真(1)．それ以外は偽(0) |
| ! | NOT | a = !b; | bが真ならばaは偽(0)．bが偽ならばaは真(1) |

〈注意〉
- ビット演算子(&, ¦, ^, ~)と間違わないこと．

**表 2-5　論理演算子**

論理演算の結果である0とか1は，そのまま算術演算に使用できます．また，逆に算術演算での算出値もそのまま論理演算と混合して使えます．この場合，算術演算の結果が0以外の値はすべて「真（1）」の扱いとなりますから注意が必要です．図2-1にその具体例を示しておきます．

なお関数も，エラーの有無などを返値(関数値)として定義してあることがあります．その場合，正常終了は0，エラーの発生は0以外などと決められています(第6章 関数を参照)．

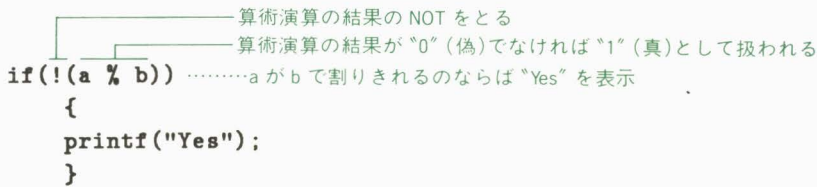
```
if(!(a % b))    ………a が b で割りきれるのならば "Yes" を表示
    {
    printf("Yes");
    }
```



図2-1　算術演算と論理演算の組合せ

## ■ その他の演算子

その他の演算子には，条件演算子とカンマ演算子があります．これらの演算子は，それほど頻繁に使われることはありませんが覚えておくと便利です．また，変数や関数のアドレス操作に使われるアドレス演算子やポインタ演算子もあります．

なお，これ以外のビット演算子とキャスト演算子については，以降の2.3節と2.4節でそれぞれくわしく取り上げます．

### ― 条件演算子(三項演算子) ―

この演算子を使うと，とくに数値計算などの例外処理を簡単に記述することができます．

表2-6に条件演算子の使用法を示します．

| 演算子 | 意味 | 使用法 |
|---|---|---|
| ? : | 条件式を作る | a = (b ? (c = d) : (c = e));<br>(c = e): b が偽(0)のとき実行<br>(c = d): b が真(0以外)のとき実行<br>b: 条件<br>a: a には c の値が入る |

〈注意〉
- この場合，とくに「()」で囲む必要はないが，見やすさのために付けてある．

表2-6　条件演算子

条件演算子は，それ1つで簡単なif文と同じ制御構造を作ってしまいます．つまり，**図2-2**のような書き換えが可能です．

〈if文での書式〉

```
if(func(fx))
    a = TRUE;
else
    a = FALSE
```

……関数funcの返値(return value)が，真(0以外)ならばaにTRUEを代入し，偽(0)ならばaにFALSEを代入する

〈条件演算子での書式〉

```
(func(fx)) ? (a = TRUE) : (a = FALSE);
```

……上記と同じ処理が簡潔に書ける

図2-2 if文と条件演算子

## — カンマ演算子(順次演算子) —

この演算子は，カンマでくくられた順に左から実行を促します．**表2-7**にカンマ演算子の使用例を示します．

| 演算子 | 意味 | 使用法 |
|---|---|---|
| , | カンマで区切られた順に左から計算を実行する | a = (a = 3, b += a, c = b * 5);<br>左から順に計算を実行していき，最終的に「c = b * 5」の結果がaに入る |
| | | for (i = 10, j = 0; i>j; i --, j ++)<br>ループの初期化とループ後の評価を2つの実行文で実行している |

表2-7 カンマ演算子

この演算子は，演算の過程を簡潔に表記したり，for文の条件などで2つの実行文を続けて書きたい場合に使用します．

## — アドレス演算子／ポインタ演算子 —

単項演算子の「&」と「*」(算術演算子と間違えないこと)は，それぞれアドレス演算子，ポインタ演算子とも呼ばれます．この2つの演算子は，演算子というよりも構文に近いものです．これらについては，「第5章 ポインタ変数」でくわしく取り上げます．

# 2.2　演算子の優先順位

C言語では演算子の数が多いので，いくつもの演算子を組み合わせて使う場合，その優先順位が気になるときがあります．通常の数値の演算では，その順位はすべて数学上の規則に則っていますが，C言語に特有の演算子をいくつか組み合わせるときには注意が必要です．

## ■ 優先順位の考え方

一般的に「演算は左側から順に行われる(左側の優先順位が高い)」,「単項演算子→二項演算子のように演算が簡単なものから実行される」という原則があります．

ただし以下のような単項演算子と代入演算子の組み合わせの例では，①では代入が先に行われ，②ではインクリメントが先に行われますので注意してください．

①a　=　b++;　……　bの値がaに代入され，その後bに1を加える
②a　=　++b;　……　bの値に1を加え，その結果がaに代入される

多くの場合，優先順位の問題は「()」(かっこ)で解決がつきます．「あ，まずいかな」と思ったり，「本当にこれで大丈夫なのかな？」という不安が残る場合はかならずかっこでくくり，演算の順位を自分で明確にしておきましょう．とくに，他の処理系に移植する可能性のあるプログラムは，そうしておかないと演算の優先順位に関する仕様が違っていることも考えられ，移植したプログラムが動かないということもあり得ます．

また逆に，かっこが多過ぎてもコンパイラが音を上げてしまう場合もあります．決まった公式に値を代入する場合を除いて，あまり複雑なかっこや式は使わないほうが賢明です．式が複雑になったときは，中間結果を別の変数に溜めておきましょう．こうするとデバッグのときにも役立ちます．とくにC言語の処理系の多くは，オーバーフロー，アンダーフローのチェックを行いませんから，こうしたデバッグに備えたプログラミングはむしろ必要なことです．

## ■ 優先順位表

C言語の演算子には，明確な優先順位があり，それらを組み合わせて使用する場合にはその優先順位に応じて演算の順序が決まります．

次ページの**表2-8**に演算子の優先順位の一覧表を示します．なお，これらの使用例は**図2-3**に示します．

| 優先順位 | 演算子の種類 | | 演算子 |
|---|---|---|---|
| 高 | 式 | | ()　[]　->　.注1 |
| ↑ | 単項演算子 | | !　~　++　--　-　(型)　*　&注2　sizeof |
| | 二項演算子 | 乗除 | *　/　% |
| | | 加減 | +　- |
| | | シフト | <<　>> |
| | | 比較 | <　<=　>　>= |
| | | 等価 | ==　!= |
| | | ビットAND | & |
| | | ビットEXOR | ^ |
| | | ビットOR | ¦ |
| | | 論理AND | && |
| | | 論理OR | ¦¦ |
| | 条件演算子 | | ?　: |
| ↓ | 代入演算子 | | =　+=　-=　*=　/=　%=　>>=　<<=　&=　^=　¦= |
| 低 | カンマ演算子 | | , |

□は，同順位(同じ列)の演算子が並んだ場合，左から右(→)に演算をする．
□は，同順位(同じ列)の演算子が並んだ場合，右から左(←)に演算をする．
このような演算の規則を**結合規則**と呼ぶ．

注1：「->」，「.」は，構造体および共用体で使われる演算子である．くわしくは第8章で取り上げる．
注2：「*(ポインタ)」,「&(アドレス)」については第5章で取り上げる．

**表 2-8　演算子の優先順位表**

---

**a=b=c=d=0;**

代入演算子は右側から順番に演算をしていくので，まずdに0が代入され，その値がcに代入され…というように実行していき，最終的にaにも0が代入される

**a>b ¦¦ c==d && e<f**

比較／等価演算子の方が論理演算子よりも優先順位が高いので，「(a>b) ¦¦ (c==d) && (e<f)」となる

***ptr++;**

ともに単項演算子なので右から順に実行する．「*(ptr++)」と同じ

**a=~b<<2;**

bの補数を左に2ビットシフトし，aに代入する．「a=(~b)<<2」と同じ

**図 2-3　優先順位の例**

# 2.3 ビット演算子

コンピュータのデータの最小単位である「ビット」を直接扱うのが，ここで紹介するビット演算子です．ビット演算子を使った演算をビット操作と呼びますが，これはC言語にあって他の高級言語にはない機能の1つです．

ビット操作は機械語を勉強した人には当り前のことなのですが，BASICとかFORTRANなどの高級言語を主にやってきた人にとっては，よくわからないこと(あるいは必要ないこと)に感じられるかもしれません．そこで以下では，C言語でビット操作がどのよう時に使われるかを中心に解説していきます．

## ■ ビットのシフト

コンピュータのなかのデータは，2進数のビット列で表現されています．まず，**図2-4**を見てください．これは2つの数値のコンピュータでの内部表現を表しています．

**図2-4　数値の内部表現**

さて，この値を2倍してみましょう．2進数の表現を頭に思い描くと，どのようにこの1バイトのなかが変わるでしょうか？　2の2倍は4ですから，それを2進数にすると**図2-5**のようになります．

また，6の4倍はどのようになるのでしょうか(**図2-6**)．

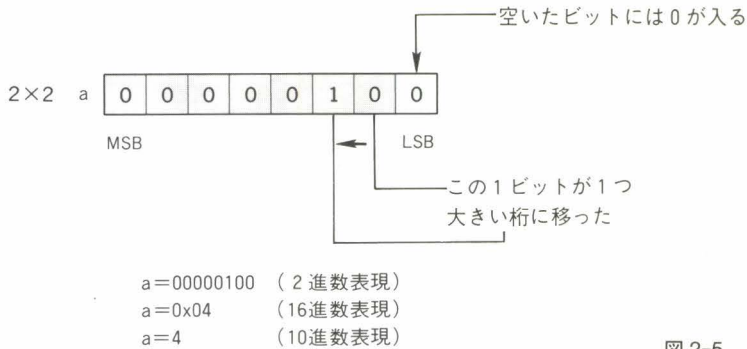



**図 2-5　4の2進数表現**

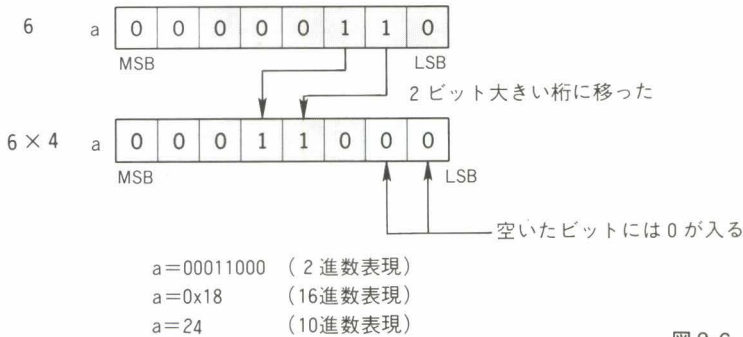



**図 2-6　24の2進数表現**

このように，ある数値を「2倍する」には，左(桁の大きい方)に1ビットずらせばよく(**シフトする**)，「4倍する」には2ビットずらせばよいのです．つまり，$2^n$の数値を掛けることはビットをn個だけ左にずらすことになります†．また逆に，$2^n$で割る場合はビットをn個右にずらせばよいことがわかるでしょう．

ところで，乗算や除算の演算子がちゃんとあるのに，なぜこのような演算が必要なのでしょうか？これは，アセンブラ(機械語)をやったことがないとちょっとわからないかもしれません．アセンブラではこのようなビットシフトと乗算の演算速度を比べると，ビットシフトのほうが数倍演算速度が速いのが普通です．今までの高級言語では演算速度を上げたいと思っても，このような計算は乗算でしかできませんでした．しかし機械語を使える場合は，ビットをシフトすることで演算の高速化が可能です．つまり，ビット操作ができるC言語でも同じように演算の高速化が計れるわけです．

† 負の数の場合は，これほど簡単でない．また，大きい数をシフトするとオーバーフローすることがある．

## ■ ビットのマスク

「ビットのシフト」が演算に使われるのに対して，「ビットのマスク」はあるビット列の特定のビットが1か0かを調べるために用います．図2-7を見てください．

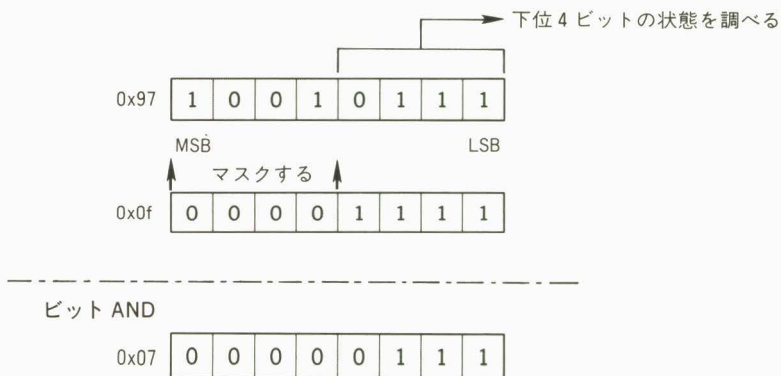



図2-7　ビットのマスク

図2-7でわかるように，「0」でANDをとった部分はマスクされ，「1」でANDをとった部分には調べたいビットと同じものが返ります．つまり，ビットANDをとることによって，特定のビットの状態を調べることができるのです．

逆にビットORをとるとどうなるでしょうか(図2-8)．

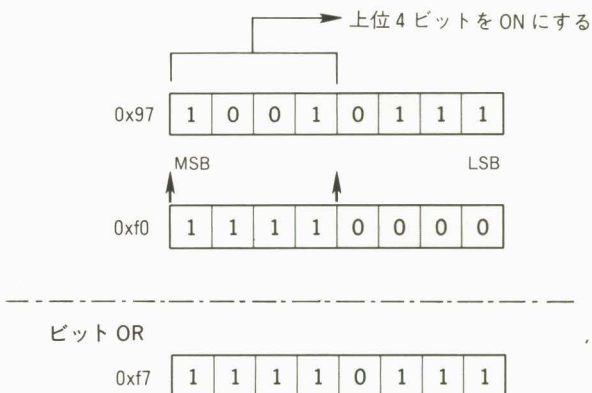



図2-8　ビットORの例

図 2-8 から,「1」で OR をとった部分が ON(1)になることがわかります. このようにビットの OR をとることで, 特定のビットを立てる(ON にする)ことが可能です.

実際に, このようなビット操作は周辺機器などをコントロールするためのフラグを調べたり, 操作したりするときに用いられます. ビット操作をプログラムで使う場合は, 以下の**図 2-9** のようにマクロ定義(第 9 章参照)をしておくとわかりやすいでしょう.

```
#define     MASK1      0x0F  ………0x0F を MASK1 として定義
      ⋮
b = a & MASK1; ………変数 a と MASK1 のビット AND をとり, 変数 b に代入
```

```
#define     MASK         0x0F          } マクロ定義に引数が使える場合は, このよう
#define     MASK1(c)     (c ¦ MASK)    } にしてもよい
      ⋮
b = MASK1(a);………変数 a と MASK のビット OR をとり, 変数 b に代入
```

**図 2-9　ビット操作の使用例**

このほかC言語には, ビットの排他的論理和やビットの反転などの演算子もあり, アセンブラと同等のビット操作ができます.

## ■ ビット演算子の種類

ビット列をシフトするシフト演算子を**表 2-9** にまとめます.

| 演算子 | 意味 | 使用法 | |
|---|---|---|---|
| << | ビットを左にシフト | a = b << 3; | b を 3 ビット左にシフトして a に代入する |
| >> | ビットを右にシフト | a = b >> 3; | b を 3 ビット右にシフトして a に代入する |

〈注意〉
- これらの演算子は, 整数 (char, int, long, およびこれらの unsigned) のみ使用できる.

**表 2-9　シフト演算子**

シフト演算子では, 右にシフトを行う場合, 符号の扱いが問題となります. 次ページの**図 2-10** に示すように, 符号付きで宣言されたデータは, 空いたビットに符号と同じ値が入り(**算術シフト**), 符号なしで宣言されたデータは 0 が入ります(**論理シフト**).

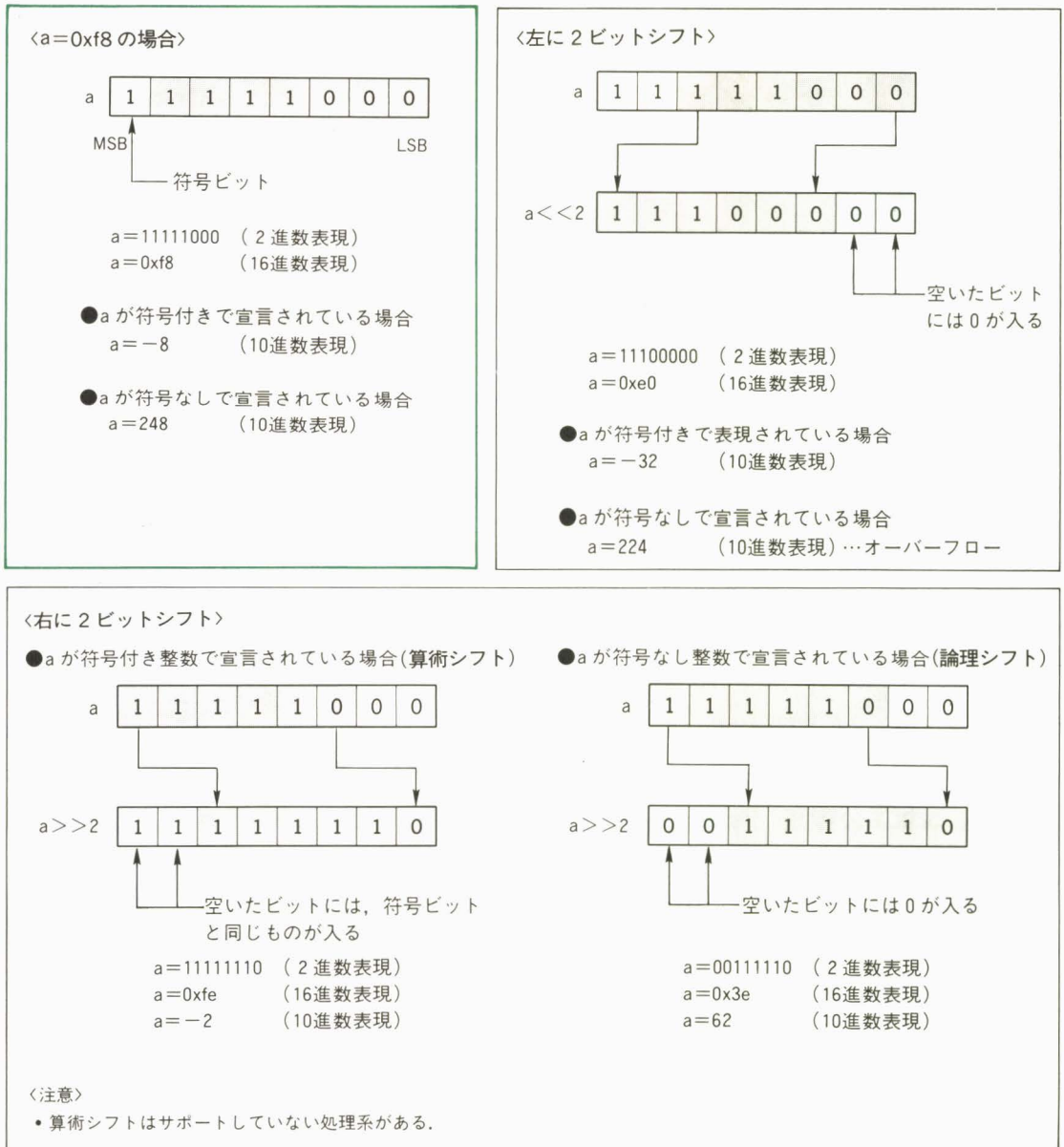



**図 2-10　算術シフトと論理シフト**

ビットのマスクなどを行うビット演算子を**表 2-10** に示します．

| 演算子 | 意味 | 使用法 | |
|---|---|---|---|
| & | ビット積 (AND) | a = b & c; | b と c のビット AND を a に代入する |
| ¦ | ビット和 (OR) | a = b ¦ c; | b と c のビット OR を a に代入する |
| ^ | ビット排他的論理和 (EXOR) | a = b ^ c; | b と c のビット EXOR を a に代入する |
| ~ | ビットの反転 | a = ~ b; | b の各ビットを反転したものを a に代入する |

〈注意〉
- 論理演算 (&&, ¦¦, ！) と間違わないこと.
- AND, OR, EXOR の関係は次に示す通り.

〈AND〉

| ＼ | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

〈OR〉

| ＼ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

〈EXOR〉

| ＼ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

**表 2-10　ビット演算子**

また，シフト演算子とビット演算子には代入演算子もあります(**表 2-11**).

| 演算子 | 意味 | 使用法 | |
|---|---|---|---|
| &= | ビット積 | a &= b; | a の値と b の値のビット AND を a に代入する |
| ¦= | ビット和 | a ¦= b; | a の値と b の値のビット OR を a に代入する |
| ^= | ビット排他的論理和 | a ^= b; | a の値と b の値のビット EXOR を a に代入する |
| <<= | 左シフト | a <<= b; | a の値を b の値だけ左にシフトし a に代入する |
| >>= | 右シフト | a >>= b; | a の値を b の値だけ右にシフトし a に代入する |

**表 2-11　ビット代入演算子**

## ■ ビット演算子を使ったプログラム

以上のようなビット演算子を使った例を掲げましょう．**リスト 2-1** のプログラムはあるアドレスのメモリを読んできて，指定されたビットが 1 か 0 かを判定するプログラムです．このプログラムの中心となる部分は関数になっていますので，他のプログラムでも そのままの形で使えます．

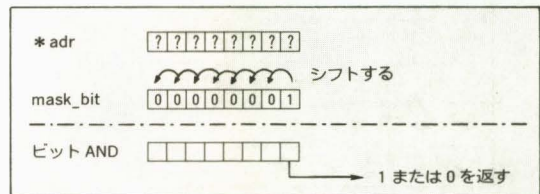
```
1: #include    <stdio.h>
2: #include    <stdlib.h>   ············atol 関数が定義されているヘッダファイルを取り込む
3:
4: #define     BOOL    int  ·········論理判断に使用する BOOL を int 型 として定義
5: #define     TRUE    1
6: #define     FALSE   0
```

```
 7:
 8: BOOL     bit_tst(adr,bit)
 9:     char     *adr;
10:     int      bit;
11:     {
12:     char     mask_bit = 0x1;
13:
14:     if(0 != (*adr & (mask_bit << bit))) return(TRUE);
15:     else                                return(FALSE);
16:     }
17:
18: main()
19:     {
20:     char     lbuff[64];
21:     char     *cadr;
22:     int      i;
23:
24:     printf(" Enter address in decimal : ");
25:     gets(lbuff);
26:     cadr = (char *)atol(lbuff);
27:
28:     printf("¥n¥n ADRRESS in HEX : %08lx   > ", cadr);
29:
30:     for(i = 7 ; i > (-1) ; --i)
31:         {
32:         if(TRUE == bit_tst(cadr,i)) printf("1");
33:         else                        printf("0");
34:         }
35:
36:     printf("¥n");
37:     }
```

返値はBOOL型（7〜8行）
指定されたアドレスとビットからビットの状態を返す関数（8行）
アドレス（9行）
ビット(0〜7)（10行）
ビットをマスクするための変数（12行）
ビットが立っていればTRUE，そうでなければFALSEを返す（14〜15行）
指定されたビット数シフトする（14行 `<< bit`）
データの取り込み（24〜25行）
データをlong型の整数に変換し，さらにそれを文字列として受け取る（26行）
キャスト演算子（次節参照）（26行 `(char *)`）
アドレスの確認（28行）
8桁のlong型の16進数，桁数が足りない場合は0で埋める（28行 `%08lx`）
ビット列の表示（32〜33行）

*adr　?|?|?|?|?|?|?|?
シフトする
mask_bit　0|0|0|0|0|0|0|1
ビットAND　 | | | | | | | 
1または0を返す

［実行結果］

```
A>test⏎
 Enter address in decimal : 12⏎


 ADRRESS in HEX : 0000000c    > 00110110

A>
```

リスト2-1　ビット判定プログラム

**リスト2-1**は，8086系CPUのスモールモデルでコンパイルすると，プログラムが実行されているセグメント内でしか動作しません．全アドレス空間にアクセスしたい場合は，ラージモデルでコンパイルしてください(スモールモデルとラージモデルについてはAPPENDIXを参照)．8ビットCPUと68000CPU等の16ビットCPU，および32ビットCPUではこのまま実行できます．

8086などのCPUでこのプログラムを使うと，システムのインタラプトベクタの値などを表示させることができ，プログラムの内部解析に使うことができます．

# 2.4 キャスト演算子

キャスト演算子は，変数のデータ型に関係する演算子でこれまで紹介した演算子とはまったく別のものです．これらは，演算子というよりも関数や構文と考えた方がわかりやすいでしょう．

ここでは，キャスト演算子と関わりが深いデータ型についても合わせて解説していきます．

## ■ sizeof 演算子

sizeof 演算子は，演算子というよりも関数と同じような働きをします．この演算子は，コンパイル時に処理系が変数に対して割り当てるメモリの大きさをバイト単位で返してきます．

**表 2-12** に sizeof 演算子の使用例を示します．

| 演算子 | 意味 | 使用法 |
|---|---|---|
| sizeof | sizeof（式［または型名］）<br><br>式または型名に対して処理系が割り当てたメモリ容量をバイト単位で返す | sizeof (256＊3)<br>計算結果は int なので，8 ビットまたは 8086 系 CPU では 2（バイト）が返る<br><br>sizeof (int)<br>8 ビットまたは 8086 系の CPU では 2（バイト）が返る |

**表 2-12　sizeof 演算子**

またこの演算子を使うと，構造体や共用体（第 8 章参照）が確保するメモリ容量を調べることもできます（**リスト 2-2**）．

この演算子は，異機種のコンピュータ同士で互換性を保つために使われます．たとえば，機種によって違いのある int 型のデータサイズや構造体が確保するバイト数を調べるときに用います．また，とくに数値のオーバーフローを気をつける必要のあるプログラムで，機種に依存しないプログラムを書きたい場合に使用することもあります．

```
1: struct meibo
2:     {
3:     char    name[50];  ………50 バイトの領域を確保
4:     int     sex;       ……… 2 バイトの領域を確保（8 ビットまたは 8086系 CPU の場合）
5:     char    tel[10];   ………10 バイトの領域を確保
6:     }                  ―――――――――――――――
                            合計 62 バイトの領域を確保
7:
```

```
 8: main()
 9:      {
10:      printf("SIZE(struct meibo)--- %d¥n", sizeof(struct meibo));
11:      }
12:
```

［実行結果］

```
A>test
SIZE(struct meibo)--- 62

A>
```

リスト 2-2　構造体が確保するサイズを調べるプログラム

## ■ データ型の変換

C言語の変数は，その属性としてデータ型を持っていますが，異なるデータ型同士で算術演算を実行すると，コンパイラは自動的に**型変換**を行います(データ型については第4章でくわしく取り上げる)．これは以下のように，式中の最も大きいデータ型に合わせて変換を行うという原則があります．

| char < short < long < float < double |
|---|

つまり, char型とlong型とdouble型の変数の演算を行う場合, すべてdouble型にデータが変換された後で計算が実行されるのです．このような演算では，かならずビット長の長い型に合わせて変換されるので精度がおちることはありません．

また，演算結果をある変数に代入する場合も，データの変換が行われます．これには，次のような原則があります．

| **左辺のデータ型に合わせて変換される** |
|---|

ここで，左辺のデータ型が右辺のデータ型よりも大きい場合は先の原則と同じく問題ありません．ところが逆に左辺のデータ型の方が小さい場合——たとえばdouble型をchar型に代入するとき，右辺のデータは自動的に切り捨てられるか丸められます(どのように丸められるかはコンパイラによって異なる)．つまり，コンパイラは警告も出さずに勝手にデータの精度をおとしてしまうのです．

コンパイラの型変換機能は，異なるデータ型を自動的に変換してくれるので非常に便利ですが，バグの原因にもなりかねませんから注意してください．最後に，データ型が自動的に変換される例を示しておきましょう(**リスト 2-3**)．

```
 1: main()
 2:     {
 3:     char     a;
 4:     short    b;
 5:     long     c;
 6:
 7:     b = 1000;     c = 100000;
 8:
 9:     printf("b + c = %d + %ld = %ld¥n", b, c, b+c);
10:
11:     a = b + c; ………計算結果は char 型に丸められる
12:     printf("b + c = %d + %ld = %d¥n", b, c, a);
13:     }

[実行結果]
A>test ⏎
b + c = 1000 + 100000 = 101000
b + c = 1000 + 100000 = -120

A>
```

このコンパイラでは下位バイトが保存される
101000 ⇨ 0x18a88
0x88 ⇨ －120

リスト 2-3　型変換の例

## ■ キャスト構文

先に述べたようにC言語では，通常の演算に際して自動的に型変換が行われますが，明示的に型変換をしたい場合は，以下の「強制的なキャスト構文」を用います．

| (データ型)式 |
|---|

この使用例を図 2-11 に示します．

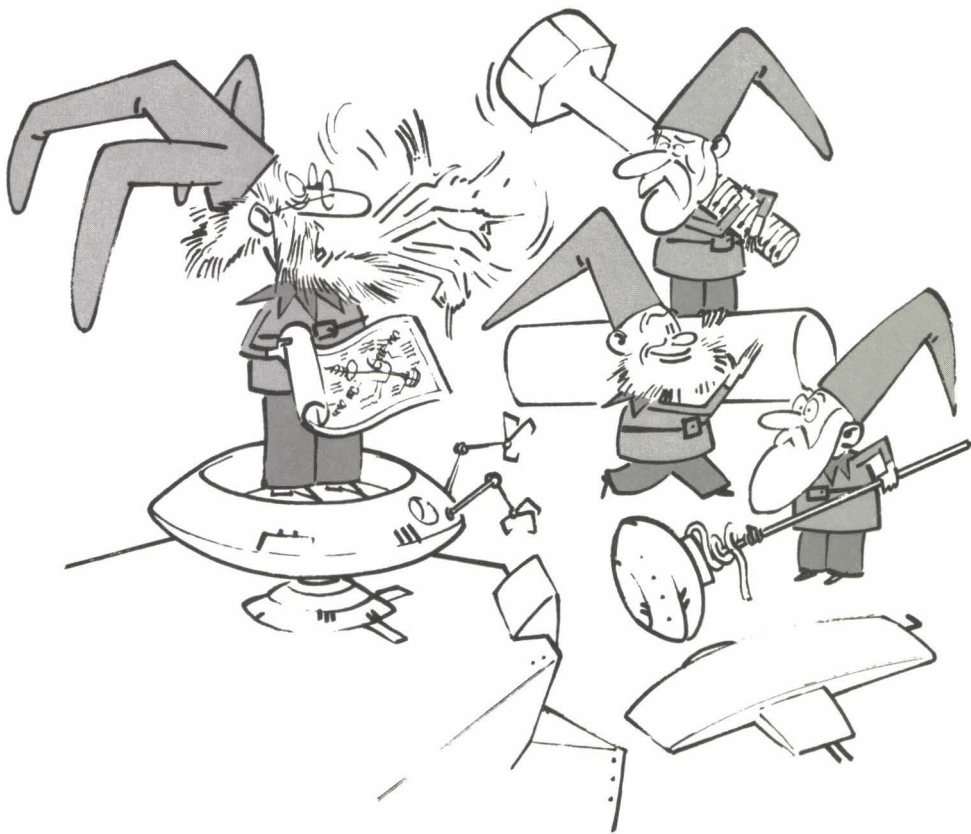
ここで，キャスト構文は値のデータ型を変換するのであって，変数の宣言を違う型に変えるわけではありません．つまり，キャストされても変数そのものは不変です．

この構文は，次ページの図 2-11 のようにデータが丸められる場合を明確に示したり，関数との間で引数や返値をやりとりする場合に用いられます．なお，キャスト構文は以降の章のサンプルプログラムでもよく使われていますので参照してください．

```
long    a;
int     c;
   ⋮
c = (int)a; ………long 型の変数が int 型に変換される
   ⋮            ことを明確に示す
```

```
func(x) ………long 型の引数を持つ関数 func の宣言
long    x;
    {

    }

main()
    {
    int     a;
     ⋮
    func((long)a); ………int 型の引数を関数 func に渡したい場合には,
     ⋮                   long 型へのキャストを行う
    }
```

図 2-11　キャスト構文の使用例

# 第3章
# 制御構造

どんなプログラミング言語にも処理の流れを記述する制御構造は存在しますが，C言語でのそれはとてもシンプルな構造をしています．制御構造としては最も一般的なif文を始め，ループを形成するwhile文，for文などがあり，またBASICのように指定した場所にジャンプするgoto文なども使うことができます．そしてこれらの制御文は，対象となる実行文を〝{}〟で囲んで明確に示すことで，その制御構造をわかりやすく記述することが可能です．

また，C言語では制御文の組み合わせがかなり自由にできますし，書式も自由に書けますから，書き方によってはかなり複雑なものができてしまいます．ここでもやはり「読みやすいプログラムを書く」という心がけが大切です．

# 3.1 制御構文

プログラムを記述する制御構造は，以下の4つに分類することができます．

① **逐次実行** …… 上から下に順番に実行していく通常の実行手順
② **条件分岐** …… 条件を評価し，その結果によって指定した実行文に分岐
③ **ループ制御** … 条件を評価し，その結果によってループを実行するかどうかを判断
④ **無条件分岐** … 指定した実行文へ無条件に分岐

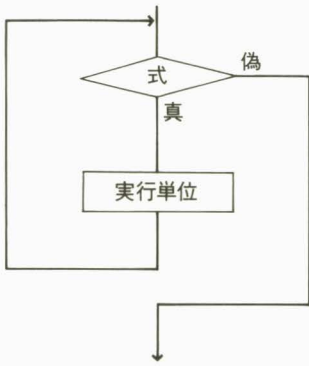
ここで①の逐次実行は，通常の実行手順ですから制御構文はありません．C言語では，以下の**表3-1**に示す制御構文を使用できます．

| | **条件分岐** | **ループ制御** | **無条件分岐** |
|---|---|---|---|
| **制御文** | if ～ else 文<br>switch 文 | while 文<br>for 文<br>do 文 | goto 文 |
| **補助文** | break 文<br>continue 文 | | ラベル文<br>return 文 |

**表3-1 制御構文の種類**

これらの制御構文の書式は，そのほとんどが関数と同じか，あるいは似かよった形をしています．次にその書式を示します．

ここで式の評価は，**0のとき偽，0以外のとき真**となりますから注意してください．以降では，**図3-1**に示した制御構文について1つずつ解説していきます．

## ■ if 文(条件分岐)

**書式1**　<if 文>

```
if (式)
  実行単位 †
```

**フローチャート1**



**書式2**　<if〜else 文>

```
if (式)
  実行単位 1
else
  実行単位 2
```

**フローチャート2**



**書式3**　<if〜else 文のネスト>

```
if (式 1)
  実行単位 1
else if (式 2)
  実行単位 2
else
  実行単位 3
```

**フローチャート3**



**使用例**

```
   ⋮
if(a > 5)
    if(b > 5)
        printf("Very Good¥n");
    else
        printf("Ok¥n");
else
    printf("Bad¥n");
   ⋮
```



**コメント**

式の評価は,「0」のとき偽,「0 以外」のとき真となる.

if 文はネストすることが可能だが，その際には「if」と「else」の対応関係に注意. else 文は，最も近い if 文から対応していくので，それ以外の組合せをする場合は「{}」で囲んで明確にその関係を示す.

† 実行単位については,「4.5 変数の有効範囲」で解説しているので参照のこと.

## ■ switch～case 文(条件分岐)

**書式 1** 〈switch 文〉

```
switch (式)
  {
  case 定数式 1 :   実行単位 1
  case 定数式 2 :   実行単位 2
       ⋮
  case 定数式 n :   実行単位 n
  default       :   実行単位 n+1
  }
```

**フローチャート 1**



**書式 2** 〈switch～break文〉

```
switch (式)
  {
  case 定数式 1 :   実行単位 1
                    break;
  case 定数式 2 :   実行単位 2
                    break;
       ⋮
  case 定数式 n :   実行単位 n
                    break;
  default       :   実行単位 n+1
  }
```

**フローチャート 2**



**使用例**

```
        ⋮
switch(ax)  ………変数 ax による分岐
    {
    case    'h':  ……………比較する定数(コロンに注意)
        printf("hello¥n");   break; ………「ax=='h'」のときに実行される実行単位
    case    'g':
        printf("goodby¥n"); break; ………「ax=='g'」のときに実行される実行単位
    default:
        printf("No. Thank you¥n"); ………上記でないときに実行される実行単位
    }
      ⋮
```

**コメント**

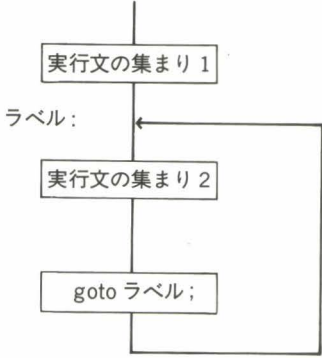
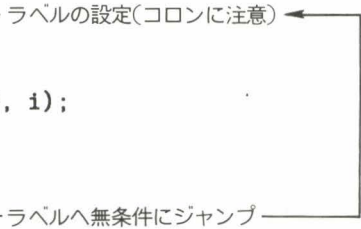
switch 文で評価される式の結果は，char 型または int 型の整数値でなければならない．また switch 文は，多くの場合 break 文とともに使われる．もし break 文がないと，どこかの case で一致した場合，その case 以降のすべての実行単位が実行されてしまうので注意．

## ■ while 文(ループ制御)

**コメント**

while 文は，ループに入る前に条件判断を行う．このとき真偽の判定は，「0 を偽」，「0 以外を真」としている点に注意．したがって，式のなかに 0 以外の数値を入れておけば無限ループができる．また，

while(1)；　　　(最後のセミコロンに注目！)

とすると，どうやっても抜けられない無限ループとなる(最後の「；」は何もしない実行文，すなわち空文である)．なお，「while()」は「while(1)」(無限ループ)と解釈される．

## ■ for 文(ループ制御)

**使用例**

```
    ⋮
for(i = 0 ; i < 10 ; ++i)  ………初期値の設定, 式の評価, 付加実行文
    {
    printf("Now is %d¥n", i);        ─for 文の実行単位
    a[i] = b[i] = i;
    }
    ⋮
```

**コメント**

for 文は, 評価式を3つ持っている. それぞれは, かっこのなかで「;」(セミコロン)で区切られる(各式は省略可能だが「;」は省略することができない). 式1は, ループに入る前に一度だけ実行される. 式2は, ループを1回まわるごとにループを抜けるかどうか判断する. 式3は, ループをまわるたびに実行される. つまり, 3つの式は以下のように呼ぶことができる.

式1:初期値設定式

式2:ループ実行判断式

式3:ループ付加実行式

このfor 文を用いると, すべてのループ構造が表現できる.

## ■ do 文(ループ制御)

**書式**

```
do
    実行単位
while (式) ;
```

**フローチャート**



**使用例**

```
        :
i = 0;
do
    {
    printf("Now is %d¥n", i);
    a[i] = b[i] = i;
    i++;
    }
while(i < 10); ………式の評価(セミコロンに注意)
        :
```



**コメント**

do 文は，while 文とは逆に実行単位を実行した後で条件判断を行う．つまり，最低1回は実行文が実行されることになる．

この制御文は，それほど使う機会は多くないが，文字列を逆順で処理したい場合などに使われる．

## ■ goto 文(無条件分岐)

**コメント**

goto 文は，ラベル文で指定した実行文に無条件で制御を移す．ラベル文は，識別子＋「：」(コロン)で構成され，通常は実行文の先頭に置かれる．goto 文は，同じ関数内でしかその効力を発揮しないので注意が必要．

この制御文は，深いネストになっているループから一気に抜け出したいときなどに使われる．ただし，このような無条件ジャンプは構造化言語のルールを崩してしまうことになるので，多用するべきではない．

# 3.2 補助制御文

return 文は，制御補助文というよりも呼ばれた関数に戻る際に使われる構文です．また，break，continue の 2 つの構文は，前述の制御構文を補助する働きをします．すなわち，単独ではなんら意味を持ちません．

## ■ return 文（関数からの復帰）

| 使用される制御構文 | すべての制御文 |
|---|---|
| 使用例 | 〈返値なし〉　〈返値あり〉 |

〈返値なし〉

```
error()
    {
    printf("Error¥n");
    return;
    }


function(ax)
    char    ax[];
    {
    int     i;
    for(i = 0 ; i < 10 ; ++i)
        if(ax[i] > 100)  error();
    }
```

〈返値あり〉

```
add(x, y)
    int    x, y;
    {
    return(x + y);
    }


function()
    {
    int    a, b, c;
    for(a = b = 0 ; a < 5 ; ++a, ++b)
        {
        c = add(a, b);
        printf("%d + %d = %d¥n", a, b, c);
        }
    }
```

**コメント**

return 文は，現在実行中の関数を抜け，その関数を呼び出した関数に制御を戻す．その際，関数が返値（return value）を持つように定義されている場合は，

```
return(式);
```

と記述し，呼び出した関数に値を返すことができる．この際，return の後のかっこは付けなくてもよいが，付けておいた方が安全である．

## ■ break 文(ループの脱出)

| 使用される制御構文 | switch 文，while 文，for 文，do 文 |
|---|---|

使用例

```
for(i = 0 ; i < 200 ; ++i)
    {
    if(a[i] == ax)
        {
        printf("Error¥n");
        break; ―――――for 文の脱出
        }
    printf("a[%d] = %d¥n", i, a[i]);
    }
  :           次の実行単位へ ←
```

コメント

break 文は，for 文などのループ構文や switch 文で使われる．この補助文は，現在実行中の実行単位から抜け出し，1 つ外側の実行単位に制御を移す．この例では，for 文の実行単位(if 文ではない)から脱出する．

## ■ continue 文(ループの続行)

| 使用される制御構文 | while 文，for 文，do 文 |
|---|---|

使用例

```
  :
for(i = 0 ; i < 200 ; ++i)
    {
    printf("a[%d] = %d¥n", i, a[i]);
    if(a[i] == ax)  continue; ――for 文の最後に制御を移す
    b[i] = a[i];
    } ←
  :
```

コメント

continue 文は，break 文とは反対に実行単位から抜け出さずに，最も内側のループ構文の最後に制御を移す．つまり，continue 文以下の処理は行わない．

C言語は，Pascalなどの他の構造化言語に比べて，実用的側面がより重視されています．そのために，数学的な側面から見ると「美しくない」構造がかなりあります．逆にいえば，実際にプログラムを組むときに楽に組めるように作られているのです．とくにC言語で組まれるアプリケーションは，実用的なものが多く，かなり広範な使用範囲を持っています．C言語は，「構造化言語」といいつつ〝goto文を持っている〟という事実もC言語の柔軟な言語構造を示しているといえるでしょう．

# 第4章
# データ型と宣言

C言語には多くの明確な「データ型」があり，「変数が扱える数値の大きさ」や「変数の有効範囲」などを決めています．このようにデータを区別しているのは，コンピュータのアーキテクチャに則したデータ型を採用することで，より高速な演算を可能にするためです．またC言語では，自分で新たなデータ型を作ることもできるため，より柔軟で多くの応用がききます．

他の言語コンパイラでも，その多くはデータ型を厳密に定義しています．そして，異なったデータ型同士での値のやり取りは非常に明確に行う必要があります．しかしC言語では，異なったデータ型の数値の演算をかなりの程度コンパイラが自動的に行ってくれます（「2.4 キャスト演算子」参照）．つまり，データ型をあまり気にせずにプログラムを組むことが可能です．

一言で言ってしまえば，C言語は「データ型の定義は厳密だが，扱いについてはあまり制限がない」言語なのです．実際のプログラムでは，とくにこのような数値の扱いがバグの原因になることも多く，コンパイラに頼らないくせをつけておくことはむしろ必要なことです．デバッグの効率を考えても，可能な限りあいまいな部分をなくすプログラミングを心がけましょう．そして，そのためにも本章をよく理解しておいてください．

# 4.1　C言語の構成要素

日本語などの私たちが使う自然言語が，あいまい性や冗長な表現を多分に許しているのに対して，C言語を含むプログラミング言語の世界では，文法やそこで使われる用語が厳密に定義されています．

ここでは，データ型について話をする前に，C言語のプログラムを構成する要素とそこで使われる用語の定義をしておきましょう．

## ■ 構成要素

C言語のプログラムは，次の7つの要素から構成されます．

① **識別子**（名前）
② **予約語**（キーワード）
③ **コメント**
④ **空白文字**（スペース，タブ，改行など）
⑤ **区切り記号**
⑥ **演算子**
⑦ **定数**

さて以下では，これらの構成要素について解説をしていきます．ただし，⑥の演算子については第2章を参照してください．また，⑦の定数については次節でくわしく取り上げます．

## ■ 識別子（名前）

識別子とは，変数，関数，構造体のタグなどに付ける名前のことです．これは次の条件を満たしている必要があります．

・使用できる文字 …… 英文字（大文字，小文字），数字，アンダースコア（_）
　　　　　　　　　　ただし，最初の文字は数字であってはならない
・識別される文字数 … 最初から8文字目まで
・予約語と一致しない（次項参照）

識別子自体はいくらでも長い名前を付けることができるので（物理的な限界はある），たとえば次のような変数名を付けることも可能です．

```
int     This_variable_is_Loop_counter ;
        | 8文字 | ——→認識されない
```

とくに最近のCコンパイラでは，8文字以上の識別子を認識してくれるものもありますが，移植性の高いプログラムを書くためには〝識別子は8文字まで〟と心えておくのがよいでしょう(とくにミニコン系のC言語は8文字までの制限が多い)．また，大文字と小文字を区別しない処理系もあり，よくプログラムを移植する際の問題となることがあるので注意してください(ただし，これはコンパイラではなくアセンブラやリンカの仕様であることが多い)．

このような変数名や関数名などの識別子は，プログラムの読みやすさやデバッグの効率にも影響してきます．あとでプログラムを見直したり他人に見られることを考えて，できるだけ明快な名前を付けるようにしてください．

なお，識別子として使われる変数や関数の宣言方法については，「4.3 データ型」以降でくわしく解説します．

## ■ 予約語(キーワード)

予約語は，C言語での用途があらかじめ定められているので，識別子として使うことはできません．これらは，以下の**表4-1**に示すようにC言語のデータ型，制御構造，演算子として使われるものです．

またいくつかの予約語は，C言語によって異なるものがあるので注意が必要です．

| データ型 | | 制御構造 | | 演算子 |
|---|---|---|---|---|
| char | auto | if | break | sizeof |
| short | static | else | continue | |
| int | extern | switch | return | |
| long | register | case | | |
| float | typedef | default | | |
| double | struct | for | | |
| unsigned | union | while | | |
| | | do | | |
| | | goto | | |

〈注意〉

• コンパイラによっては，const, enum, void, entry, asm, fortran なども予約語となっているものがある．

**表4-1　C言語の予約語**

## ■ コメント

「/*」と「*/」で囲まれた部分は，コメントとして認識されコンパイルの対象となりません．コメントは識別子や予約語を区切ってしまわない限りどこにでも書くことができ，また複数行のコメントも可能です．

ただし，**図4-1**のようなコメントのネスト(入れ子)を許す処理系とそうでない処理系がありますので注意してください．

```
        :
/*
int     function(a, b)
int     a, b;
        {
        return( a > b ? a : b);  /* max value */
        }
*/
        :
```

プログラムをデバッグするためにこの関数をコメントで囲って実行しないようにすると，コメントのネストでエラーとなる処理系がある

**図4-1 コメントのネスト**

## ■ 空白文字

スペース，タブ，改行(Carriage Return)，改ページ記号は，空白文字と呼ばれプログラム中の識別子，予約語を分離する区切り記号となります．また，識別子などを区切るために任意の個数の空白文字を入れることができるので，行にとらわれない自由なフォーマットでプログラムを書くことができます(これを**フリーフォーマット**という)．前述で示したコメントなどは，この空白文字と同様に扱われます．

例外として，#include，#defineなどのプリプロセッサに与える指示は，行を単位としています．また，通常「#」は1カラム目から始まる必要があります(プリプロセッサについては，第9章でくわしく解説している)．

```
#include
      <stdio.h>
```

#include ……… 2行にわたって書くことはできない

通常は1カラム目から書く

## ■ 区切り記号

**表4-2**に示す記号類は，C言語のプログラムのなかでその構造を明確にする目的で使われます．これらの記号はプログラムの骨格をつなぐリベットのようなもので，文法上のエラーはこの記号類の抜けに起因することが多いようです．とくに，1つの実行文の区切りを示す「；」(セミコロン)と複数の実行文のまとまり(ブロック)を示す「{…}」には注意してください．

なお，キャラクタコード「0x5C」はJISコードでは「¥」(円記号)がASCIIコードでは「＼」(バックスラッシュ)が割り当てられています．

| 記号 | 読み | 意味 | 使用例 |
|---|---|---|---|
| ; | セミコロン | 1つの実行文 | int counter ;<br>for (i = 0 ; i < 5 ; i++) … |
| : | コロン | ラベル | case 1 : … |
| {…} | 大かっこ | 複文(実行単位) | for (i = 0 ; i < 5 ; i++)<br>{<br>a [i] = b [i] ;<br>⋮<br>} |
| (…) | かっこ | 制御文の条件<br>関数の引数 | if ( a > b) …<br>main (argc, argv) … |
| [...] | 角かっこ | 配列の要素数 | char a [10] ; |
| <…> | 鋭角かっこ | #include 文で取り込むファイル | #include <stdio. h> |
| ' | シングルクォーテーション | 1文字 | a = '¥n' ; |
| " | ダブルクォーテーション | 文字列<br>#include 文で取り込むファイル | static char a [] = "ABC" ;<br>#include "ioctl.h" |
| ¥<br>\ | 円記号 注<br>バックスラッシュ | エスケープキャラクタ<br>(画面の制御など) | printf ("test¥n") ; |
| % | パーセント | エスケープキャラクタ<br>(表現指示文字列) | printf ("%d¥n", c) ; |

注：JISコードでは¥(円記号)になり，ASCIIコードでは\(バックスラッシュ)となる．

**表4-2　C言語で使われる区切り記号**

# 4.2　定数

定数とは，プログラム中に直接値を書き込み，プログラム実行中にその値が書き変わることのない数値です．変数として宣言したものでも，扱いによっては定数だという呼び方もありますが，実際のプログラムでは「宣言なしで使われる値が常に一定した数」ということになります．以下の表現の右辺はすべて定数です．

```
a = 0x41;
a = 'A';
a = 65;
    :
```

（a = 0x41; a = 'A'; a = 65; について）定数には代入ができないので，左辺にくることはない

このような定数の表現には，**表 4-3** に示す 4 種類があります．このうち，文字列を使った表現だけが若干他のものと扱いが異なります．

| 表現 | | 例 | 備考 |
|---|---|---|---|
| 整数 | 10 進 | 14 | —— |
| | 16 進 | 0x14 | 先頭に「0x」を付ける |
| | 8 進 | 014 | 先頭に「0」を付ける |
| 浮動小数点数 | | 3.1415 | E を付けて指数部を表現することも可能 |
| 文字 | 文字 | 'A' | シングルクォーテーションで囲む |
| | 制御文字 | '¥n' | エスケープキャラクタとともに使う |
| 文字列 | | "STRINGS" | ダブルクォーテーションで囲む |

**表 4-3　定数の表現**

これらの表現について，以下で解説していきましょう．

## ■ 整数表現

整数の表現には，以下の**表 4-4** に示す 10 進，16 進，8 進表現の 3 種類があります．

| 整数 | 使用する数値 | 表現例 | コメント |
|---|---|---|---|
| 10 進数 | 0 ～ 9 | 256 | —— |
| 16 進数 | 0 ～ 9，A ～ F（小文字も可） | 0x100 | 先頭に 0（ゼロ）と x（小文字のエックス）を付ける |
| 8 進数 | 0 ～ 7 | 0400 | 先頭に 0 を付ける |

**表 4-4　整数の表現**

これらの負の整数は，数学の表記と同様に先頭に「－(マイナス)」を付けて表現します．

また各数値は，その大きさに合わせて「int」あるいは「long」の型が自動的に指定されますが(「4.3 データ型」を参照)，明示的に long 型を指定したい場合は数値列の最後に「L(l)」を付けて示します．

```
a=100L;  ……… 10 進数の 100 を明示的に long 型で指定する
```

## ■ 浮動小数点数表現

符号付きの実数を 10 進数で表現するのが，この浮動小数点数です．その書式は以下のようになっています．

| [符号]整数部[．小数部][E[符号]指数部] |
|---|

このうち，整数部か小数部のいずれかは省略可能です．指数部を指定する「E」は小文字を使うこともできます．次にその指定例を示します．

浮動小数点数の型は，すべて double 型が指定されます．また，小数点形式で指定されたデータは，内部で指数表示に変換されます(「4.3 データ型」を参照)．

## ■ 文字表現

ある単一の文字や制御文字は，「'」(シングルクォーテーション)で囲んで表現されます．文字データは，内部的には数値(アスキーコード)として与えられるので，たとえば次のような数値演算を行うことも可能です(どの文字がいくつのアスキーコードに対応しているかは前巻のAPPENDIXのキャラクタコード表を見てください)．また文字定数の型は，char型になります(「4.3 データ型」参照)．

```
c = 'A' +0x20;  ……… 変数cに小文字の「a」を代入する
```

制御文字は，画面のコントロールやそのままでは直接表現できない文字を与えるために用いられます．表4-5に制御文字の一覧表を示します．

| 制御文字 | 機　能 |
|---|---|
| ¥n | 復帰と改行(改行し，次の行の先頭にいく) |
| ¥t | 水平タブ |
| ¥b | 1文字分戻る |
| ¥r | 同じ行の先頭に戻る |
| ¥f | ページ送り，またはスクリーンクリア |
| ¥¥ | 円記号そのものを表す |
| ¥' | シングルクォーテーションを表す |
| ¥" | ダブルクォーテーションを表す |
| ¥002 | 8進数で指定したアスキー文字(頭の0は省略してもよい) |
| ¥x10 | 16進数で指定したアスキー文字(頭は小文字のxで，残りは16進数の2桁) |

〈注意〉
• ここで挙げたすべての制御文字をサポートしていない処理系もある．

**表4-5　制御文字**

これらの制御文字は，単独の文字としてだけでなく，次に解説する文字列のなかでも以下のように使うことができます．

```
printf("ABCDEF¥"GHIJ");  ……「ABCDEF"GHIJ」という文字列の表示
```

## ■ 文字列表現

文字列は，「"」（ダブルクォーテーション）で囲まれて表現されます．C言語には文字列というデータ型が存在しないので，文字列は文字の配列として扱われます．

C言語で文字列を表記すると，1文字ごとに配列に格納され最後に**ヌル文字**(¥0)が自動的に付加されます．ヌル文字は，コンパイラが文字列の終わりを認識するためのものです．**図4-2**に，文字列の内部表現を示します．

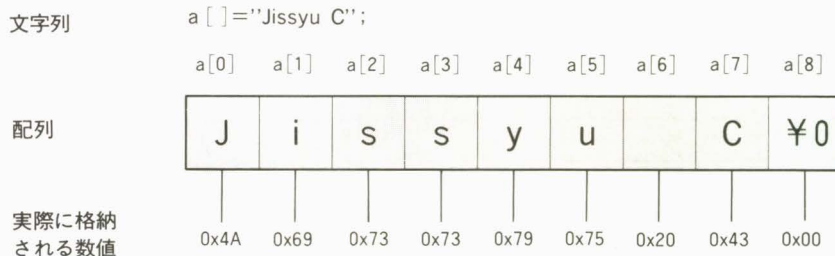



**図4-2　文字列の内部表現**

なお，2行以上に分けて文字列を表現したい場合は，文字列の最後にエスケープキャラクタ(¥)を付けることで可能になります．

"You say YES, I say NO.¥
You say STOP, but I say GO! GO!"

という表記は，

"You say YES, I say NO. You say STOP, but I say GO! GO!"

と同じです．

文字列についての扱いは，「第5章 ポインタ変数」のところでくわしく解説していますので参照してください．

# 4.3　データ型

識別子として使われる変数は，前節で解説した定数と違い特定の値を持ちません．変数は，「初期化」あるいは「代入」という手段(演算)によって値が与えられます．関数は，返値(関数値)を持つように定義されている場合，変数と同様に扱うことができます(くわしくは第6章を参照)．また，このほかに特殊な変数として構造体，共用体がありますが，これについては第8章で解説します．

C言語では，変数や関数などを使う場合，その属性をあらかじめ宣言しておかなければなりません．C言語の変数の属性には，以下の3種類があります．

**[属性1]**　表現できる数の大きさ(データ型)
**[属性2]**　変数のメモリ上の位置(記憶クラス)
**[属性3]**　プログラム上のどの場所で使えるか(有効範囲)

この節では，[属性1]のデータ型について解説していきます．[属性2]の記憶クラスについては4.4節で，[属性3]の有効範囲については4.5節で，それぞれくわしく取り上げます．

## ■ データ型の種類

変数には**表4-6**に示す「型」があり，そのデータ型で表せる数値の範囲にそれぞれ制限があります．

| 種別 | 符号の有無 | ビット長 | 表現 | 数値の範囲 |
|---|---|---|---|---|
| 整数 | あり | 8 | char | −128〜＋127 |
| | | 16 | short | −32,768〜＋32,767 |
| | | 32 | long | −2,147,483,648〜＋2,147,483,647 |
| | なし | 8 | unsigned char | 0〜255 |
| | | 16 | unsigned short | 0〜65,535 |
| | | 32 | unsigned long | 0〜4,294,967,295 |
| 浮動小数点数 | あり | 32 | float | およそ $10^{-38}$〜$10^{38}$(10進数最大7桁の精度) |
| | | 64 | double | およそ $10^{-306}$〜$10^{306}$(10進数最大16桁の精度) |

〈注意〉
- C言語の処理系により，表現が異なるものやサポートしていないデータ型があるので，使用に際してはマニュアルを参照のこと．
- 浮動小数点の精度は，使用するハードウェアに依存する．

**表4-6　データ型の種類**

**表4-6**に示したデータ型以外に，各CPUのアーキテクチャに依存した**int型**があります．int型は，「どのCPUでも動くプログラムを作る」ことを目標にしてプログラムの構築をする場合，かならず問題が起きる項目の1つです．以下の**表4-7**を参照の上，十分注意したプログラミングが望まれます．

| CPU | z80，8080などの8ビット系<br>8086，V30などの16ビット系 | 68000などの16ビット系<br>32ビット系CPU |
|---|---|---|
| int型 | shortと同じ | longと同じ |

**表4-7　CPUによるint型の違い**

また，ここで紹介する基本的な算術型を組み合わせて作られるデータ型(複合型)として，**配列型**，**関数型**，**ポインタ型**，**構造体**，**共用体**がありますが，これらについては本章以外の各章でそれぞれくわしく解説します．

## ■ 符号付きと符号なし

**表4-6**で示したように，変数にはそれぞれ表せる値の範囲があります．とくに気をつけなければならないのは，整数の場合「符号付き」と「符号なし」の2種類の同じビット長の整数があるということです．この選択を間違えると，「0から増やしているはずの数がループしている最中にマイナスの値になってしまう」といった，奇怪な現象が頻発することになります．以下の**図4-3**などもよくある勘違いです．

```
main()
    {
    short     i;  ………変数iを16ビット符号付き整数として宣言する
      ⋮
    for(i = 0 ; i < 0xFFFF ; ++i) ………0xFFFFは負の数なのだが…
        {
          ⋮

    }
```

**図4-3　符号の勘違い**

また，標準関数を使用する場合に，そこで使われる引数や返値の型の間違いもよくあります．このような誤りは，多くのコンパイラではエラーメッセージとして指摘してくれないので注意してください．

## ■ ビット長

整数の場合，正負の数の扱いなどでビット長が問題となることがあります．以下の**図 4-4** に整数型データの内部構造を示しておきます．

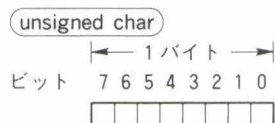
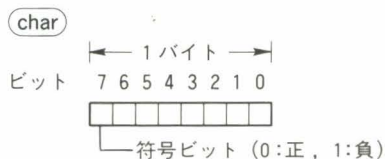
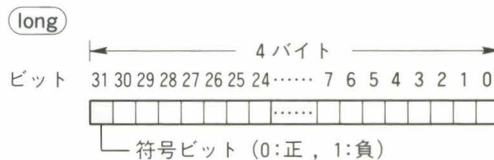



**図 4-4 整数型データの内部構造**

## ■ 浮動小数点数[†]

float と double は，それぞれ仮数部と指数部を持つ浮動小数点数です．double は float の 2 倍の精度（2 倍長の仮数部）を持っています．**図 4-5** に float と double 型のデータの内部構造を示します．

† ここで解説する浮動小数点数の内部構造と精度は，ハードウェアによって異なるので注意する．

浮動小数点型の書式

±1.［仮数部］×2$^{[指数部]}$ ……先頭に「1.」が付くと仮定する

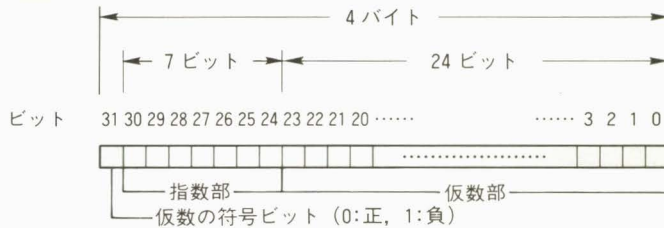
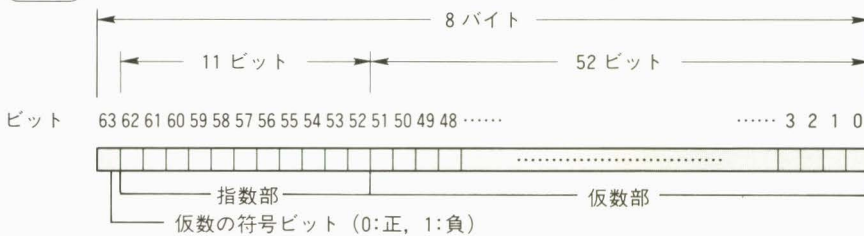
float

double

**図4-5　浮動小数点型データの内部構造**

floatとdoubleでは，計算の丸め誤差に違いがでてきます．次の**リスト4-1**を実行して，その精度を確認してみましょう．

**リスト4-1**に示すように，float型では計算誤差がでてしまいました．ただし，通常の演算ではよほど必要がない限りdouble型を使うことはないでしょう．double型は，たとえばグラフィック関係のプログラムでレイ・トレーシングを行う場合や科学技術計算で単純精度が必要になる場合に使われるのが普通です．

また，C言語をこれらの計算に使うようになったのはごく最近のことなので，浮動小数点数を使った計算に関する環境はなかなか揃っていないというのが現状のようです．コンパイラのバグがこの辺に潜んでいる例も少なくありません．

```
 1: main()
 2:     {
 3:     int    i;
 4:
 5:     float  x;
 6:     double y;
 7:
 8:     x = y = 0.0;  ………浮動小数点数であることを明示するために小数点以下を付けている
 9:
10:     for(i = 0 ; i < 10000; i++) ………float と double で宣言した変数について,
11:         {                                10 万を 1 万回加算する
12:         x += 100000.0;
13:         y += 100000.0;
14:         }
15:
16:     printf("float  ----> %f¥n", x / 100000.0); } 計算結果を 10 万で割る
17:     printf("double ----> %f¥n", y / 100000.0); }
18:     }
```

[実行結果]

```
A>test
float  ----> 9998.517760 ………float の場合,この処理系では約 0.015% の誤差が出た
double ----> 10000.000000 ………double の場合 誤差はなし

A>
```

リスト 4-1 計算精度を確認するプログラム

## ■ データ型の変換

異なるデータ型同士の演算は，コンパイラが自動的に優先順位の高いデータ型に合わせて変換し計算を行います(「2.4 キャスト演算子」参照)．その優先順位を次に示します(右側の方が優先順位が高い)．

| char < short < long < float < double |
|---|

ただし代入演算子を使っている場合は，計算結果が左辺のデータ型に変換されます．その際に，ビット長の短いデータ型に変換されると切り捨てや丸め誤差が生じますので注意が必要です(どのように丸められるかはコンパイラによって異なる)．また，強制的に型を変換したり，型変換を明示するときには，**キャスト演算子**を用いることができます(「2.4 キャスト演算子」参照)．

# 4.4 記憶クラス

変数はメモリ上にその領域が確保されますが，そのときに割り当てられる大きさを指示するのが前節で解説したデータ型です．これに対して，ここで解説する記憶クラスは，メモリ上のどの位置に変数を割り当てるのかを決定します．

この節では，記憶クラスの種類とその使い方についてまとめていきます．なお，記憶クラスと密接な関係にある変数の有効範囲については，次節で取り上げます．

## ■ 記憶クラスの種類

C言語では，その変数の属性として「メモリ上のどこ置くか」という記憶クラスを決めておかなければなりません．**表4-8**に記憶クラスの種類を示します．

| 種類 | 記憶場所 | 意味 | 宣言の例 |
|---|---|---|---|
| auto | スタック<br>(メモリ上の一時的な記憶領域) | 宣言された実行単位内でのみ有効でその実行単位を抜けた後は，その値は保存されない | int a;<br>(通常，記憶クラスの宣言は省略される) |
| register | CPUのレジスタ | auto変数と同じだが，使用頻度が高く，高速化をはかりたい場合に使用する | register int a; |
| static | 通常のメモリ | 宣言された実行単位内でのみ有効でその実行単位を抜けた後も，その値は保存される | static int a; |
| extern | ―― | 別のコンパイル単位で宣言されたグローバルな同名の変数の使用を宣言する | extern int a; |
| typedef | ―― | 新しいデータ型を作成する | typedef int BOOL; |

〈注意〉
• typedef変数は，サポートされていない処理系もある．

**表4-8　記憶クラスの種類**

この表のなかで，「実行単位」，「コンパイル単位」という用語がでてきますが，これについては次節で解説します．

## ■ auto 変数

auto 変数は**スタック領域**に記憶される変数です．スタックとは，LIFO(Last In First Out)のバッファ領域で，記憶した順序とは逆にデータを取り出すことができます．以下の**図 4-6** にスタックの原理を示します．

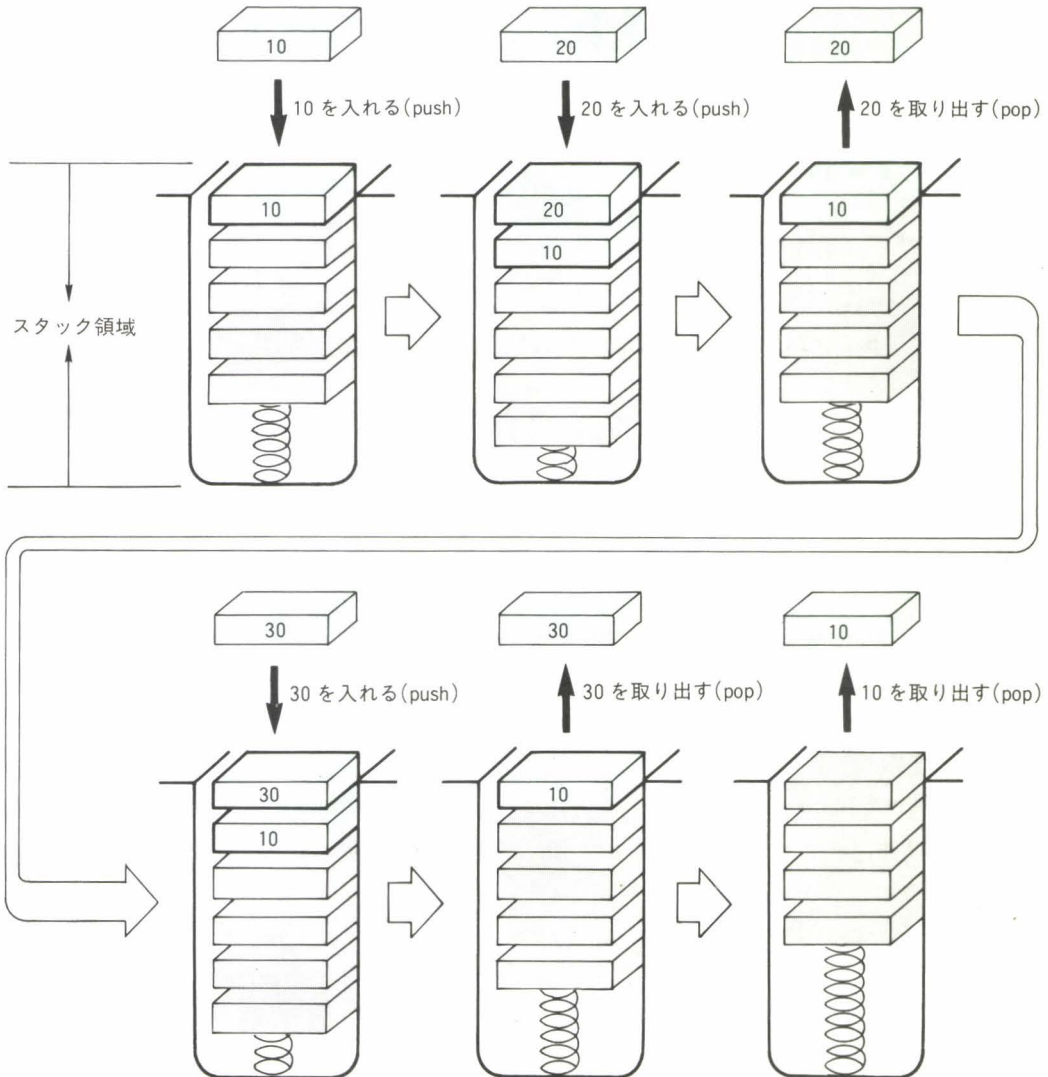



**図 4-6　スタックの原理**

図4-6のように，スタックは一時的に変数の値を蓄えておく共用の記憶領域です．つまり，特定の実行単位でのみ使用する変数は，auto変数としておくことでメモリの有効利用がはかれ，また実行単位ごとの変数の独立も保つことができるわけです．図4-7にプログラムの実行とスタックの動作をまとめておきます．

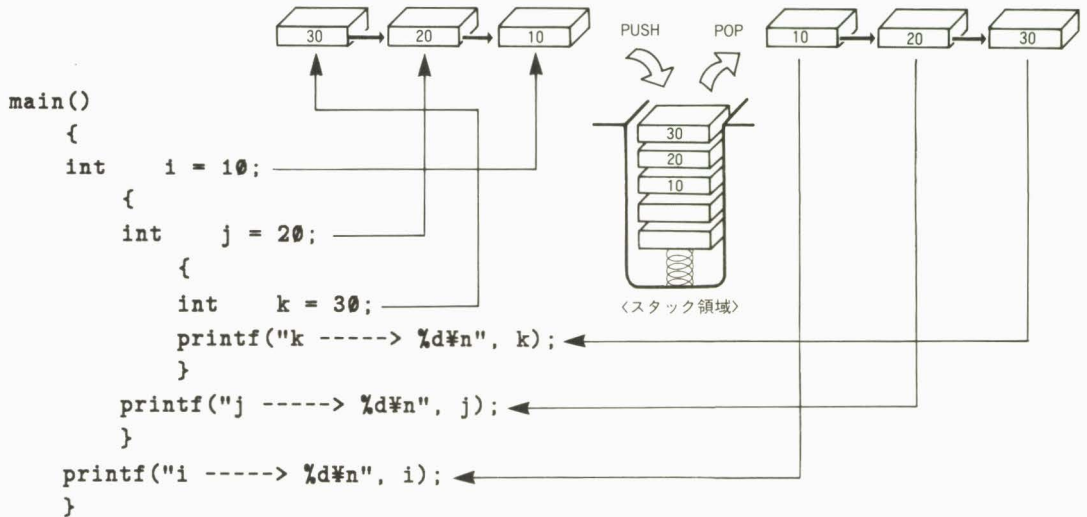



図4-7　プログラムの実行とスタックの動作

なお, autoという記憶クラスの宣言は通常省略されます(また, 省略しないといけない処理系もあります).

## ■ register変数

register変数はCPUのレジスタに記憶される変数で，とくに使用頻度の高い変数を宣言しておくことで演算の高速化をはかることができます．ビット操作などと同じく，この宣言もC言語のアセンブラ的な機能の1つといえるでしょう．

ただし, CPUによって使用できるレジスタの個数は異なりますから, 無制限にregister宣言できるわけではありません(コンパイラがレジスタに割り当てられなかった変数はauto変数と同じ扱いになる)．8086CPUなどでは，通常2つ程度が目安となります(図4-8)．

またデータ型としては，char，short，int，およびポインタのみ宣言可能で，long，doubleなどは使えません．さらにregister変数のアドレスを求めることはできません(&演算子を使用できない)．

なお，次節で解説する変数の有効範囲は，auto変数と同じ扱いになります．

〈注意〉
・ここでは，Microsoft C Compiler の register 変数で使われる8086CPUのレジスタを示した．
・8086CPUのレジスタ構成については，APPENDIXの「8086系CPUの概説」(281ページ)を参照のこと．

ソース・インデックス

ディスティネーション・インデックス

図 4-8 8086CPU のレジスタ構成

## ■ static 変数

static 変数は，プログラムと同じように通常のメモリ領域に割り当てられます．つまり，スタックと違いコンパイラが一度記憶場所を割り当てると，常にその場所に値を保存しておくことができます．このため，多くの実行単位で共通に使われる変数や，実行単位を抜けた後も値を記憶しておく変数として使われます．

以下の図 4-9 に static 変数とメモリの関係をまとめます．

```
static char    name[] = "ABC";

main()
    {
        :
    for(i = 0 ; i < 4 ; i++)
        printf("name[%d] -----> %c¥n", i, name[i]);
        :
    }

sub()
    {
        :
    printf("name -----> %s¥n", name);
        :
    }
```

name[0]
name[1]
name[2]
name[3]
A
B
C
0
〈通常のメモリ領域〉

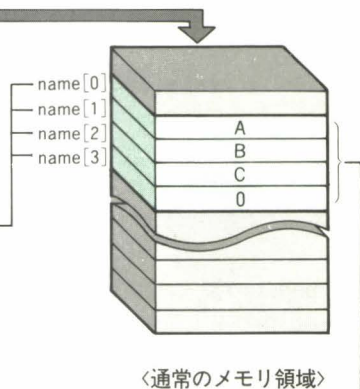



図 4-9 static 変数とメモリ上の位置

## ■ extern 変数

この変数は，メモリ上に記憶領域を割り当てるわけではありません．分割コンパイル(第9章で解説)で，別のソースファイルにある変数を参照したい場合にその使用宣言をするためのものです．この変数については，次節でくわしく解説します．

## ■ typedef 変数

この変数はこれまで解説した変数とは異なり，新たなデータ型を作成するときに用いられるものです．たとえば，図4-10のように定義した構造体(第8章参照)を新しいデータ型として定義したり，既存のデータ型を別の名前で定義することができます．

```
struct person ………構造体 person の宣言
    {
    char     name[60];
    char     address[100];
    char     tel[20];
    int      sex;
    };          以後，構造体 person を新たなデータ
                型 PERSON として使用する
typedef struct person     PERSON;

PERSON    data[100]; ………PERSON 型の配列
                           data の使用宣言
```

```
typedef char     *strings; ……
                     (char  * )型を新たなデータ型
main()               strings として定義
    {
    strings     a; ………strings 型を使った宣言

    a = "Nyumon -> Jissyu -> Ouyou";

    }
```

図4-10　typedef の使用例

typedefの書式は以下の通りです．

| typedef　現存するデータ型　新たに定義したデータ型； |
|---|

なお，typedefのない処理系では，プリプロセッサの「# define」(第9章参照)を代わりに使う場合もあります．

# 4.5 変数の有効範囲

C言語の変数の宣言は，前述のデータ型と記憶クラスの2つを使って行われます．ここで，その変数がプログラム上のどこで宣言されるのかという問題は，変数の有効範囲と関連し構造化言語として非常に重要な意味があります．

## ■ 宣言の位置と変数の有効範囲

変数は宣言される位置によって，**ローカル変数**と**グローバル変数**の2種類に大別できます．また，ローカル変数かグローバル変数であるかによって，使用できる記憶クラスも異なります．まずこれらを表でまとめておきましょう(**表4-9**)．

| 種別 | 宣言される位置 | 使用できる記憶クラス | 意味 |
|---|---|---|---|
| ローカル変数<br>(局所変数) | 特定の実行単位の中 | auto, register<br>static | 宣言された実行単位内でのみ有効 |
| グローバル変数<br>(外部変数) | 関数の外 | static<br>extern | すべての実行単位で有効 |

**表4-9　ローカル変数とグローバル変数**

## ■ 実行単位とコンパイル単位

**表4-9**の実行単位という用語について解説します.「実行単位」というあまり聞き慣れない言葉は，筆者の造語です．C言語では，その構造の区切りとして4.1節で解説した区切り記号を用いますが，このうち最も重要なのは「；」と「{}」です．「；」はシーケンシャルな1つの実行文を区切るのに使われ，「{}」は複数の実行文を1つの実行文として扱うために使われます．この「{」と「}」で囲まれた範囲を実行単位といいますが，この記号によってC言語はその構造を明確にすることができるのです(**図4-11**)．

また，先の**表4-8**にある「コンパイル単位」とは，コンパイラが一度にコンパイルする1つのソースファイルのことです．ちょっと大きなプログラムを書いていくと，かならず機能別にモジュール化を進めていくことになりますが，そこで使われるのが**分割コンパイル**という手法です(第9章参照)．つまり，**図4-12**に示すように個々のソースファイルごとに別々にコンパイルを行い，その結果できたオブジェクトファイルをリンカで1つのプログラムにまとめるのです．

```
main()
    {
    int     i; ………1つの実行文                      main( )の実行単位
     :
    for(i = 0 ; i < 100 ; i++)
        {                                            for 文の実行単位
        if((c = getc(fp)) == EOF)
            {                                        if 文の実行単位
            a[i] = b[i];
            break; ………break 文は，for 文の実行単位を抜ける
            }
        putc(c, stdout);
        }
     :
```

図4-11　実行単位

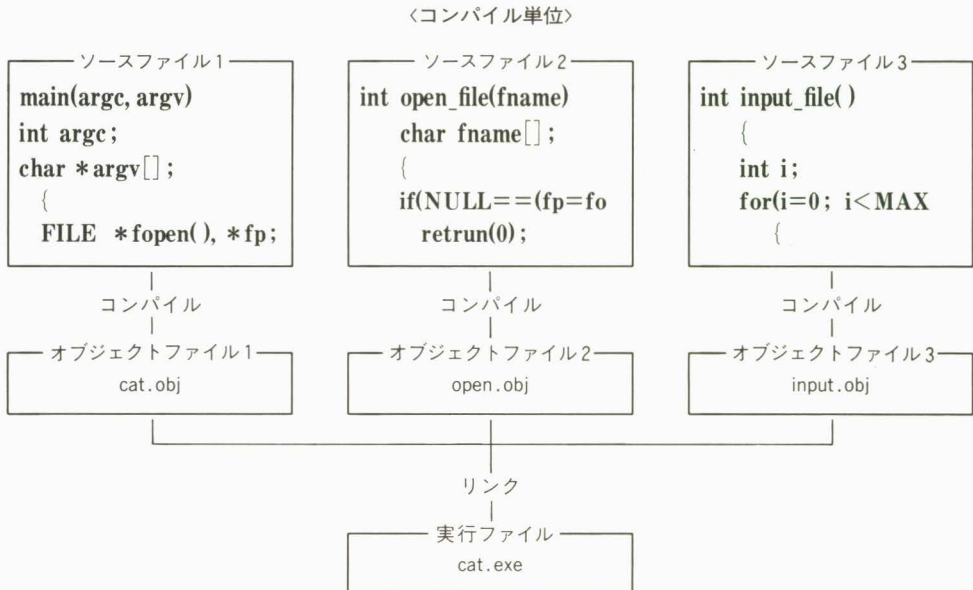



図4-12　分割コンパイル

ここで問題となるのは，各コンパイル単位間での変数の引き渡しです．これには前節の extern 変数を用います（くわしくは以降で取り上げます）．

## ■ ローカル変数

ローカル変数は「局所変数」とも呼ばれ，宣言された実行単位内でのみ有効です．記憶クラスとしては，auto で宣言される場合と static で宣言される場合があります．

まず，auto で宣言される場合を考えてみましょう(**リスト 4-2**)．

```
 1: main()
 2:     {
 3:     int i;                              変数iの有効範囲
 4:     for(i = 0 ; i < 2 ; i++)
 5:         {
 6:         int i;                       変数iの有効範囲
 7:         for(i = 0 ; i < 3 ; i++)
 8:             {
 9:             int i = 0;            変数iの有効範囲
10:             printf("i(No.3) ---> %d¥n", ++i);
11:             }
12:         printf("i(No.2) ---> %d¥n", i);
13:         }
14:     printf("i(No.1) ---> %d¥n", i);
15:     }
```

[実行結果]

```
A>test
i(No.3) ---> 1
i(No.3) ---> 1
i(No.3) ---> 1
i(No.2) ---> 3      ― 9行目の変数iの値は，ループが実行されるたびに初期化される
i(No.3) ---> 1
i(No.3) ---> 1
i(No.3) ---> 1
i(No.2) ---> 3
i(No.1) ---> 2

A>
```

**リスト 4-2　auto で宣言されたローカル変数**

**リスト 4-2** に示すように，「{}」のなかで宣言されたローカル変数は，その実行単位でのみ使用することができます．そして，異なる実行単位では同名の変数も区別されます．auto 変数はスタックによる一時記憶なので，宣言された実行単位を抜けるとその値は消えてしまい，変数の初期化はその実行単位に入るごとに毎回行われます．なお，関数の仮引数も，auto で宣言されたローカル変数として扱われます．

これに対して static で宣言された場合は，主記憶上に変数領域が取られますから実行単位を抜けてもその値は保持されます．また変数の初期化は，コンパイル時に一度だけ行われます．前述の**リスト4-2**の9行目の auto 変数を static 変数に変更して結果を確かめてみましょう(**リスト4-3**)．

```
 1: main()
 2:     {
 3:     int i;
 4:     for(i = 0 ; i < 2 ; i++)
 5:         {
 6:         int i;
 7:         for(i = 0 ; i < 3 ; i++)
 8:             {
 9:             static int i = 0; ………static 変数で宣言し0で初期化する
10:             printf("i(No.3) ---> %d¥n", ++i);
11:             }
12:         printf("i(No.2) ---> %d¥n", i);
13:         }
14:     printf("i(No.1) ---> %d¥n", i);
15:     }
```

[実行結果]

```
A>test
i(No.3) ---> 1
i(No.3) ---> 2
i(No.3) ---> 3
i(No.2) ---> 3
i(No.3) ---> 4
i(No.3) ---> 5
i(No.3) ---> 6
i(No.2) ---> 3
i(No.1) ---> 2

A>
```

9行目の変数 i はコンパイル時に初期化され，実行時は保存される

**リスト 4-3　static で宣言されたローカル変数**

## ■ グローバル変数

グローバル変数は「外部変数」とも呼ばれ，すべての実行単位で有効な変数です．この変数はローカル変数と異なり関数の外で宣言され，記憶クラスは static, extern, および宣言しない場合の3通りがあります．

まず，static で宣言された場合を見てみましょう(**リスト4-4**)．

```
 1: static int i; ………変数 i はどの実行単位からでも参照できる
 2:
 3: main()
 4:     {
 5:     i = 3; ………グローバル変数に値を代入
 6:         {
 7:         int i; ………ローカル変数とグローバル変数が衝突した場合は，ローカル変数が優先される
 8:         for(i = 0 ; i < 2 ; i++)
 9:             printf("i(main) ---> %d¥n", i); ………ローカル変数の表示
10:         }
11:
12:     printf("i(main) ---> %d¥n", i); ………グローバル変数の表示
13:     sub();
14:     }
15:
16: static int j = 9; ………変数 j はこれ以降の実行単位でしか参照できない
17:
18: sub()
19:     {
20:     i += 3; ………グローバル変数に値を代入
21:
22:     printf("i(sub)  ---> %d¥n", i); ┐
23:     printf("j(sub)  ---> %d¥n", j); ┘ グローバル変数の表示
24:     }
```

［実行結果］

```
A>test
i(main) ---> 0 ┐
i(main) ---> 1 ┘ ローカル変数の表示
i(main) ---> 3 ┐
i(sub)  ---> 6 │ グローバル変数の表示
j(sub)  ---> 9 ┘

A>
```

**リスト 4-4　static で宣言されたグローバル変数**

**リスト4-4** に示すように，関数の外で宣言されたグローバル変数は，宣言された位置以降のすべての実行単位で使用することができます．ここでグローバル変数と同名なローカル変数が存在した場合は，ローカル変数が優先されます．また static 変数なので，コンパイル時に初期化することができます．

これに対して extern 変数は，別のコンパイル単位にあるグローバル変数を利用することを宣言するものです．extern 変数のあるコンパイル単位には変数の実体はないのですが，別のコンパイル単位に存在する同名のグローバル変数とリンクされることによって実行可能になります．具体的な実行例は第 9 章で示しますので，ここではその概念図だけを**図 4-13** に示しておきます．

cat.c

```
static char     file_name[40];   … cat.cファイルのすべての実行単位で使える
                                   配列 file_name の宣言
main(argc, argv)
int   argc;
char *argv[];
  {
  FILE *fp;
  ⋮
```

open.c

```
int open_file()
  {
  extern char   file_name[];   ……………………… open_file 関数内で cat.c
                                              ファイルの配列 file_name
  if(NULL == (fp=fopen(file_name, "r")))     を使用することを宣言
      return(0);                             (特定の関数内でしか参照
  else                                       できないローカル変数と
  ⋮                                          しても使える)
```

input.c

```
extern char   file_name[];   … input.c ファイルのすべての実行単位で
                               cat.c ファイルの配列 file_name を使用す
int input_file()               ることを宣言
  {
  int i;
  for(i=0; i<MAXFILES; ++i)
      {
  ⋮
```

→ 実行ファイル cat.exe

〈注意〉
• コンパイラによっては，グローバル変数を参照する場合，extern 宣言を省略できるものもある．

図 4-13　extern 変数の使用例

図 4-13 にあるように, extern 変数をある実行単位のなかで宣言した場合は，ローカル変数のように参照範囲をその実行単位にのみ限定することもできます．

記憶クラスを宣言しない場合は，リンカによって externとして扱われるか static 領域に変数が確保されるかが決まります．つまり，他のソースファイルに同名の変数がなければ0に初期化され，同

名の変数がある場合はexternとして扱われます．**図4-14**に記憶クラスの宣言を省略した場合の例を示しておきます．

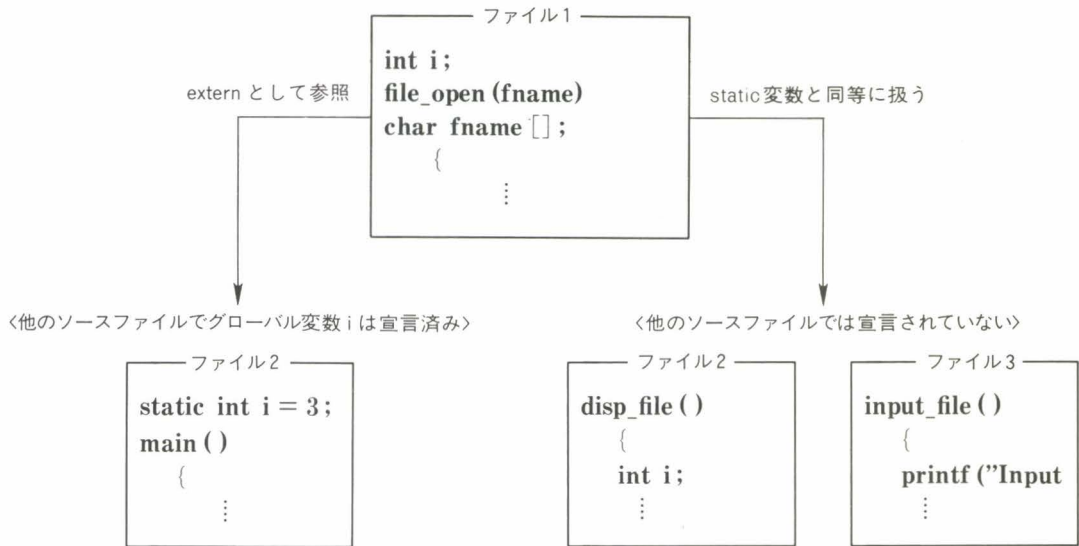



**図4-14　記憶クラスを省略した場合のグローバル変数の扱い**

また，異なるソースファイルで同名のグローバル変数を別々に使いたい場合は，かならずstaticで宣言しておく必要があります(**図4-15**)．

ソースファイル1と2のグローバル変数sizeは，まったく別の変数として認識される

ソースファイル1

```
static char size [5];
main (argc, argv)
int  argc;
char *argv [];
    {
        ⋮
```

ソースファイル2

```
static char size [5];
sub (n, m)
int n, m;
    {
    char name [40];
        ⋮
```

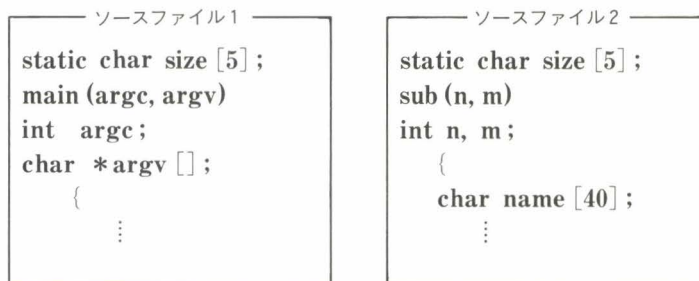



**図4-15　異なるソースファイルで同名のグローバル変数を使う**

## ■ 関数の宣言と有効範囲

関数の宣言は基本的に変数の場合と同様ですが，関数の有効範囲という点については若干異なります．関数は一度定義されると，どのコンパイル単位からでも呼び出すことができます．関数の呼び出しでは，通常記憶クラスが省略されますが，これは変数でいえば extern に当たります（もちろん，外部関数であることを明示するために extern を付けることもできる）．また，特定のコンパイル単位でのみ使用する関数であるあることを示すためには，static で宣言します．これは，異なるコンパイル単位で同名の関数を使いたい場合などに使用します．

図 4-16 に関数の宣言についての例を挙げます．

ソースファイル1
```
main()
  {
     :
  a = sub 1 ();              ……関数呼び出し（extern を省略）
     :
  extern long sub 2 ();      ……関数の使用宣言
     :                           （外部関数であることを明示するため
  }                              に extern を使うことも可能）
static min_max ()            ……内部関数の定義
  {                              （ソースファイル 1 のみで使用可能）
     :
  }
```

ソースファイル2
```
static min_max ()            ……内部関数の定義
  {                              （ソースファイル 1 の min_max 関数とは
     :                           区別される）
  }
int  sub 1 ()                ……関数の定義
  {
     :
  }
long sub 2 ()                ……関数の定義
  {
     :
  }
  :
```

分割コンパイル ── 実行ファイル

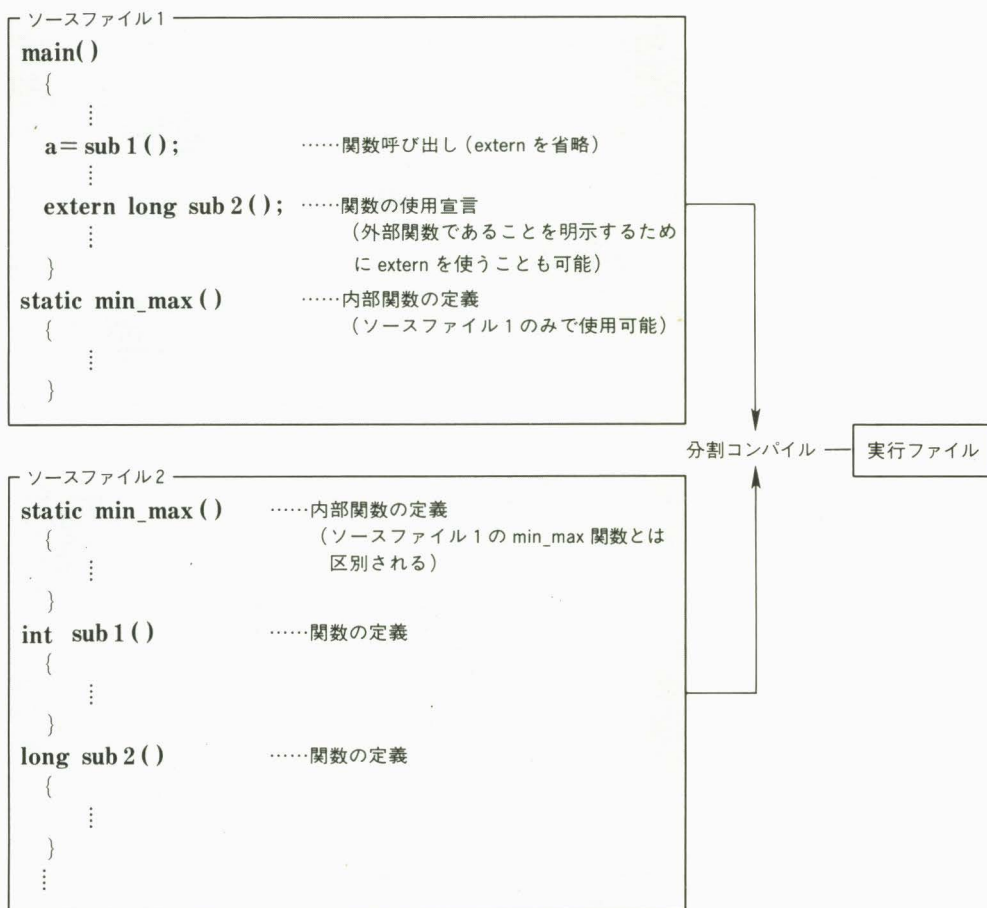



図 4-16　関数の宣言

# 4.6　宣言方法のまとめ

ここでは最後に，これまできちんと述べてこなかった初期化についての話と，データ型や記憶クラスによる宣言方法のまとめをしておくことにします．

## ■ 初期化

変数の宣言に続けて初期値を明示することによって，使用する変数をあらかじめ初期化しておくことができます．ただし，変数がどの記憶クラスに属するかによって多少その意味が異なりますので，まずそれを表にしておきましょう(**表 4-10**)．

| 記憶クラス | 初期化のタイミング | 初期値 | 初期値が省略された場合 | 配列，構造体，共用体の初期化 |
|---|---|---|---|---|
| auto<br>register | 宣言された実行単位に制御が移るたびに毎回初期化される | 定数値／変数値ともに可能 | 値は不定 | 不可能 |
| static | コンパイル時に一度だけ初期化される | 定数値のみ可能 | 0 で初期化 | 可能 |
| extern | 初期化は不可能 | | | |

**表 4-10　変数の記憶クラスと初期化の関係**

以下の**表 4-11** に初期化の例を示しておきます．

| 分類 | 表記の例 | 解説 |
|---|---|---|
| 数値 | static int a = 12 ;<br>int b = (a * 2) ; | auto 変数は変数を使った初期化も可能 |
| 文字 | char c = 'S' ; | 1 文字はシングルクォーテーションで囲む |
| 配列 | static int a[3] = {12, 24, 36} ;<br>static char a[2][2] = { {'a', 'b'}, {'c', 'd'} } ;<br>static int a[][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} } ; | 配列は「 {} 」で囲んで初期化を明示する．また，2 次元配列の先頭の要素の数は省略することも可能 |
| 文字列 | static char c[4] = "ERR" ;<br>static char c[] = "ERR" ;<br>static char *c = "ERR" ; | 文字列はダブルクォーテーションで囲み，1 次元配列として初期化する．文字列の最後にはヌル文字(¥0) が自動的に付加される |

〈注意〉
• 配列と文字列については第 5 章でくわしく解説する．

**表 4-11　初期化の例**

初期化は，一般に次の書式で行います．

| **変数の宣言 ＝ 初期値(または式)；** |
|---|

前ページの**表 4-11** について簡単に解説しておきましょう(配列と文字列については第 5 章でくわしく解説する)．

配列の初期化は，「{}」で囲まれた間にカンマで区切って要素を並べます．2 次元配列の場合は，**表 4-11** に示したように要素のまとまりごとに「{}」で区切ります．ただし，これは見やすさのためだけなので，次のように書いてもかまいません．

```
static char    a[2][2] = {'a', 'b', 'c', 'd'} ;
```

この場合，a[0][0]，a[0][1]，a[1][0]，a[1][1]の順に「{}」のなかの要素が順番に代入されていきます．また，配列で指定した要素数よりも「{}」内の要素の数が少ない場合は 0 で初期化されます．

文字列による初期化は，char 型の 1 次元配列で行います．文字列の最後には自動的にヌル文字(¥0)が付加されますから，配列の要素数はかならず(文字数＋1)を指定する必要があります(配列の要素数は省略することもできる)．

## ■ 宣言方法のまとめ

変数や関数の宣言は，これまで解説したデータ型，記憶クラス，初期値を使って次の書式で宣言を行います．

| **[記憶クラス]　データ型　変数名(配列名，関数名…)　[＝初期値][，変数名…]；**<br>[]は省略可能 |
|---|

**図 4-17** に宣言の例を示します．

```
int    i, *ptr, *sub();
```

変数 i は int 型，*ptr と関数sub の返値は int 型のポインタ変数で宣言(記憶クラスは auto)

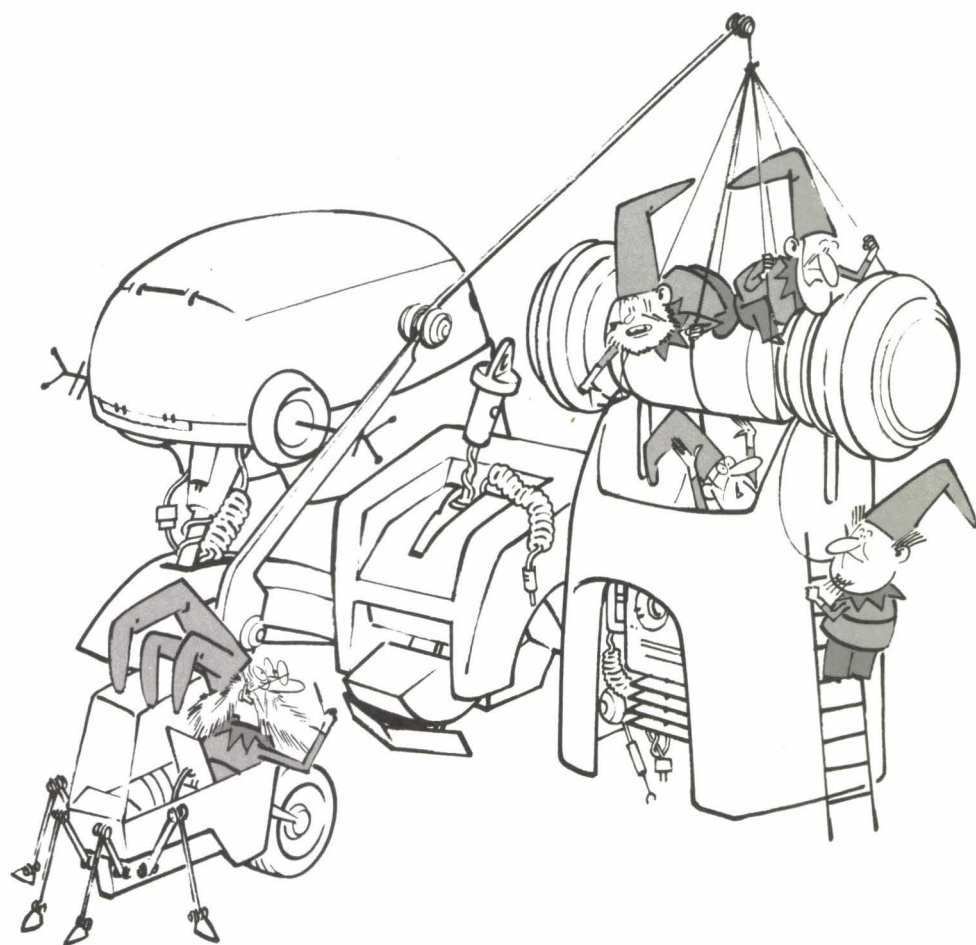
```
static char    *a, b[]="Hello";
```

変数 a は char 型のポインタ変数，配列 b は char 型で宣言し，配列 b を初期化する(記憶クラスは static)

```
long    a=(x*5), *b=&d, (*p)[10];
```

変数 a は long 型，*b と配列 *p は long 型のポインタ変数で宣言し，変数 a と変数 *b を初期化する(記憶クラスは auto)

**図 4-17　宣言の例**

# 第5章
# ポインタ変数

C言語のプログラミングに当たって最初に引っかかる「難関」はなんといっても，「ポインタ変数」の扱いに尽きるようです．

ポインタ変数は，値そのものを扱う普通の変数とはかなり勝手が違います．普通の変数がコンピュータの世界とコンピュータ以外の世界との関わりを作るものであるとすると，ポインタ変数はコンピュータの世界のなかで自己完結してしまう変数であるといえるでしょう．そのため，わからない人にとってはいっこうにわからないという，まるで宗教のような性質があります．つまり，その表現と実体がなかなか一致しないのです．このポインタ変数を乗り越えるには，その表現と実体を一致させる努力をする以外にありません．本章の説明は，この点に注意して読んでみましょう．

# 5.1　1次元配列とポインタ変数

配列とポインタ変数は非常に密接な関係があり，配列の操作はポインタ変数を使うことで同じように行うことができます．そこでここでは，1次元配列とポインタ変数の関係を探ってみることにしましょう．実際のプログラムでよく使う配列変数やポインタ変数は，その半分以上が文字列の処理を中心としていますので，ここでは文字配列とポインタ変数の話を中心に進めていきます．

## ■ ポインタ変数

変数はメモリ上にその領域が確保され，コンパイラ(正確にはコンパイラのランタイムルーチン群)によって管理されていますが，C言語ではその変数のアドレスを積極的に利用できるように考え出された変数があります．このようなアドレスを管理できる変数のことを**ポインタ変数**と呼びます．

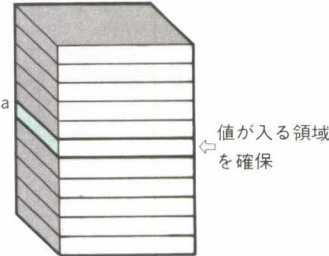
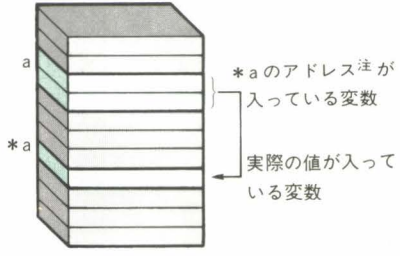
ポインタ変数は，すべての基本データ型(第4章を参照)に対して用いることができます．char型の変数の場合の「通常の変数」と「ポインタ変数」の違いを以下の**表5-1**に示してみましょう．

| | 通常の変数 | ポインタ変数 |
|---|---|---|
| 宣言方法 | char　a; | char　*a;　(単項演算子「*」を使う) |
| 変数への値の代入 | a ='A'; | *a = 'A'; |
| 変数のアドレスの参照 | x = &a;　(単項演算子「&」を使う) | x = a; |
| アドレスの操作 | 不可能　例: &a = 3500; | 可能　例: a = 3500; |
| アドレスの演算 | 不可能　例: x = &(a + 1)　; | 可能　例: x = a ++; |
| メモリ上の領域 | a<br>値が入る領域を確保 | a<br>*a<br>*aのアドレス注が入っている変数<br>実際の値が入っている変数<br>注:ここではアドレスを16ビットで想定 |

**表5-1　通常の変数とポインタ変数の違い**

ポインタ変数の宣言は,

```
char   *a;
```

というように変数名の前に単項演算子「*」を付けます.この場合,以下の2つの変数が宣言されると考えればよいのです.

| |
|---|
| *a;　……char型の変数<br>a;　……ポインタ変数(*aのアドレスを表す変数) |

実際,コンパイラは内部的に2つの領域を確保します.また次のように,aにはアドレスを代入できるので,*aで参照できるデータは代入されたアドレスにあるメモリの値となります.

```
a = 0x21BF;
```

つまり,C言語では「CPUのアクセスできるアドレスならば,どんなところでも読み書きできる」ということになります.そして,この機能こそC言語の構造化アセンブラとしての真髄なのです.**図5-1**にポインタ変数の概念図を示しておきます.

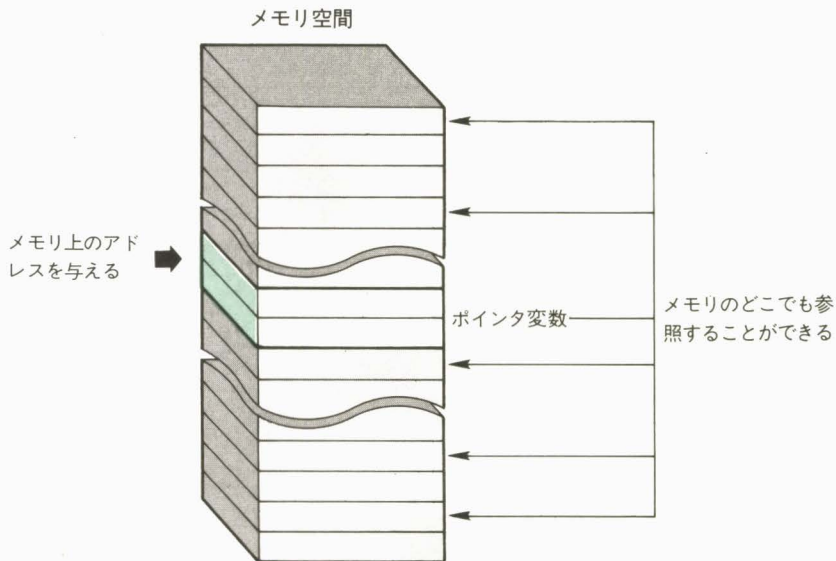



**図5-1　ポインタ変数の概念図**

通常の変数が，その変数のアドレスを直接アクセスするのに対して，ポインタ変数では，まずアドレスの入っている変数を調べ，そのアドレスで示される変数の値を読みにいきます．このように間接的に変数を指し示すポインタ変数は，以降で解説する配列を扱う上で絶大な威力を発揮します．

ポインタ変数には，もう1つ大きな利点があります．それは，通常のC言語の変数を修飾するデータ型であるということです．図5-2を見てください．

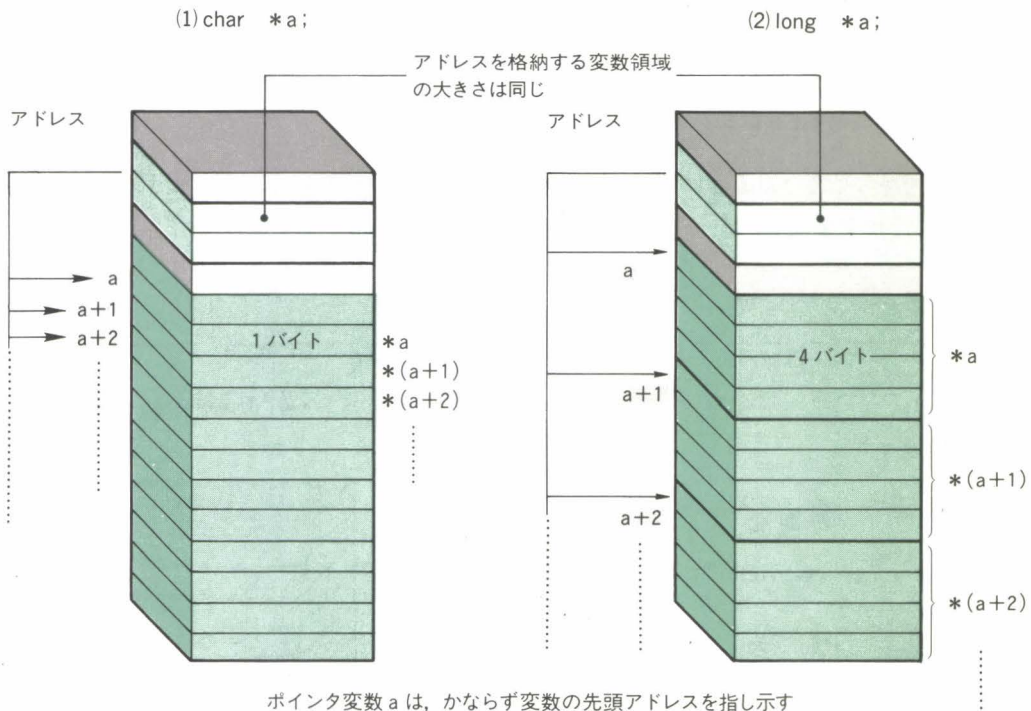



図5-2 char型とlong型のポインタ変数

図5-2でわかるように，アドレスを指し示す変数aをインクリメントした場合，char型で宣言されたポインタ変数では1バイトごとに，long型で宣言されたポインタ変数では4バイトごとにアクセスが行われます．つまり，プログラマはデータ型を指定するだけで，具体的なアドレスの操作をいっさい気にする必要はありません．もし，int型をlong型に変えたければ，最初の宣言を変更すればすんでしまうのです．

このようにC言語では，ポインタ変数を使うことでアドレスという物理的な実体を通常の変数と同じように簡単に扱うことができます．

## ■ 1次元配列

C言語での1次元配列は，次のように宣言されます．

| 記憶クラス　データ型　配列名[定数式][，配列名[定数式]……]； |
|---|

宣言された配列は，メモリ上の連続した領域として確保されます．また，初期値を設定する場合は，以下のようになります(第4章参照)．

**……　配列名[定数式] ＝ 〝文字列〞；**
**……　配列名[定数式] ＝ {数値，数値，……}；**

まず，文字列から見ていくことにしましょう．C言語では，文字列をchar型の配列として扱います(他の言語のように文字列というデータ型はありません)．その宣言と内部構造は，以下の**図5-3**のようになります．

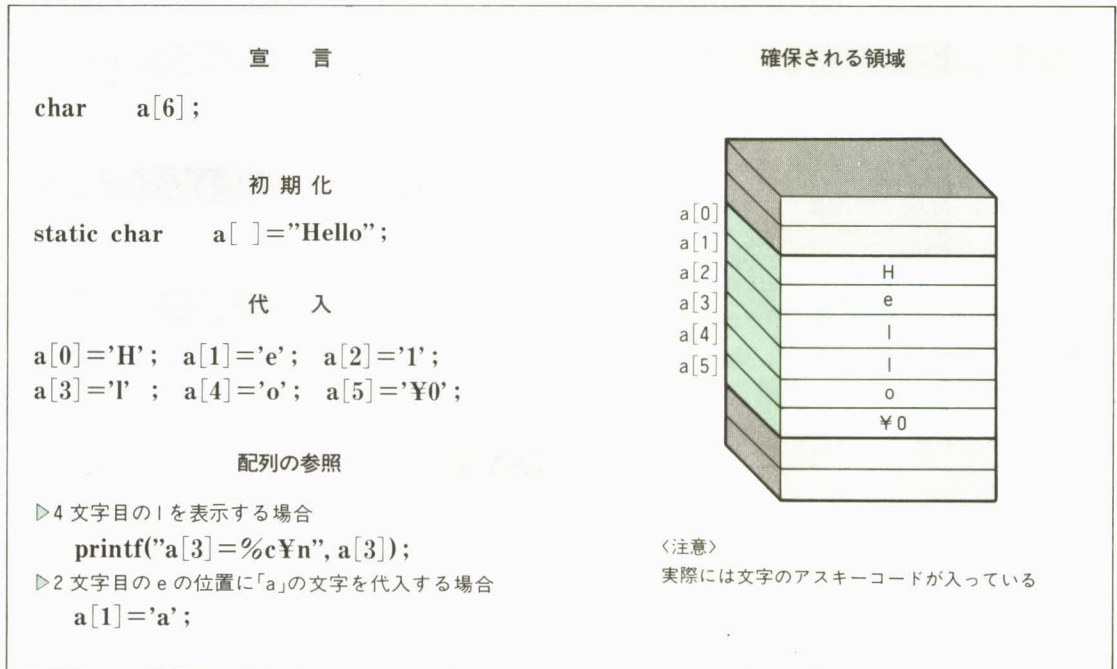



図5-3　文字列と配列

ここで，配列の添字は0から始まるという点に注意してください(a[6]であれば，a[0]～a[5]まで)．また配列の要素数は，その配列が初期化される場合か関数の仮引数として宣言される場合のみ省略することが可能です．

C言語での文字列は，その終わりにかならず**ヌル文字**(¥0)が必要ですが，文字列の初期化を行った場合はコンパイラによって自動的に付加されます．つまり，**図5-3**に示すように(文字数＋1)の領域が確保されることになります．要素数の指定を自分で行う場合には，このヌル文字の存在を忘れないようにしてください．

数値の配列の場合は，**図5-4**のようになります．

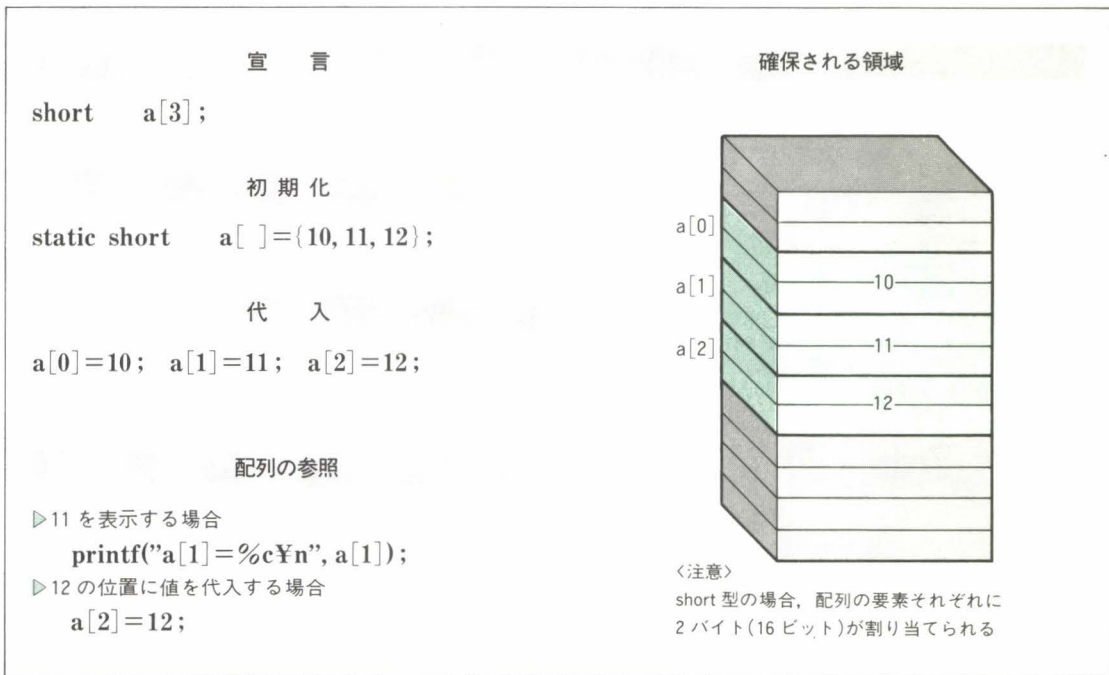



**図5-4　数値と配列**

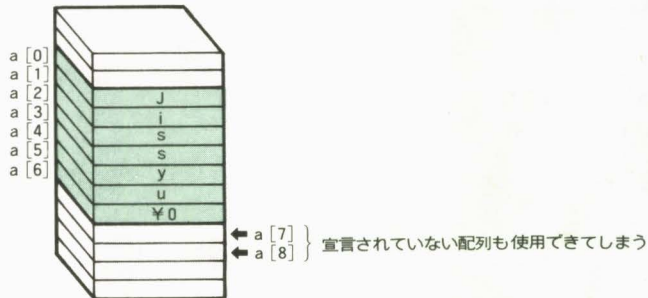
数値配列は，ヌル文字が付かない点を除けば文字列と同じです．ただし，配列の要素1つ1つが確保するバイト数は，宣言されたデータ型によって異なります(第4章参照)．

C言語で配列を扱う場合，最も気をつけなければいけないのは，コンパイラが添字のチェックを行わないという点です．**リスト5-1**の例を見てみましょう．

```
 1: main()
 2:     {
 3:     char    a[7];  ………a[0]～a[6]までの7つの要素を持つchar型の配列aの宣言
 4:     int     i;
 5:
 6:     a[0] = 'J', a[1] = 'i', a[2] = 's';  ………配列の1つ1つに文字を代入していく
 7:     a[3] = 's', a[4] = 'y', a[5] = 'u', a[6] = '¥0';  ……文字列の最後には，ヌル文字を代入
 8:                                                        （初期化する場合と違い，自動的に
 9:     for(i = 0 ; i < 8 ; i++)                            挿入されない）
10:         printf("a[%d] -----> %c¥n", i, a[i]);  ………宣言された要素数を越えて読み出しを行う
11:
12:     a[6] = ' ';
13:     a[7] = 'C';   } 宣言された要素数を越えて書き込みを行う
14:     a[8] = '¥0';
15:
16:     printf("¥na[]  -----> %s¥n", a);
17:     }
```

［実行結果］

```
A>test ↵
a[0] -----> J
a[1] -----> i
a[2] -----> s
a[3] -----> s
a[4] -----> y
a[5] -----> u
a[6] ----->        ………画面上には表示されないが，実際にはヌル文字(¥0)が入っている
a[7] ----->        ………a[7]は宣言されていないが，読み出しができてしまう
                       （画面上に表示されない文字が入っていたようだ）
a[]  -----> Jissyu C  ………宣言した配列の領域を越えて，書き込みもできる
                         （別の変数領域を壊してしまった可能性が強い）
A>
```

a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
J
i
s
s
y
u
¥0
← a[7]
← a[8]
宣言されていない配列も使用できてしまう

リスト5-1　添字のチェック

この例でわかるように，C言語では宣言した配列の要素数を越えて読み出しや書き込みができてしまうのです．最悪の場合は，別の変数領域を壊してしまうことにもなりかねません．このような間違いは，実行時の暴走という結果に即結びつきますから，配列を扱う際にはとくに注意してください．

## ■ ポインタ変数と文字列

文字列は，配列だけでなくポインタ変数を使っても同様に扱うことができます．まず，ポインタ変数を使った文字列の使用宣言を見てみましょう（図 5-5）．

宣　言

```
char    *a;
```

初 期 化

```
char    *a="Hello";
```

代　入

```
a="Hello";
```

最初の文字 'H' を格納するアドレスを変数 a にセット

配列の参照

▷4 文字目の l を表示する場合

```
printf("*(a+3)=%c¥n", *(a+3));
```

▷2 文字目の e の位置に「a」の文字を代入する場合

```
*(a+1)='a';
```

確保される領域

| | |
|---|---|
| | a |
| | |
| H | *a |
| e | *(a+1) |
| l | *(a+2) |
| l | *(a+3) |
| o | *(a+4) |
| ¥0 | *(a+5) |

〈注意〉
実際には文字のアスキーコードが入っている

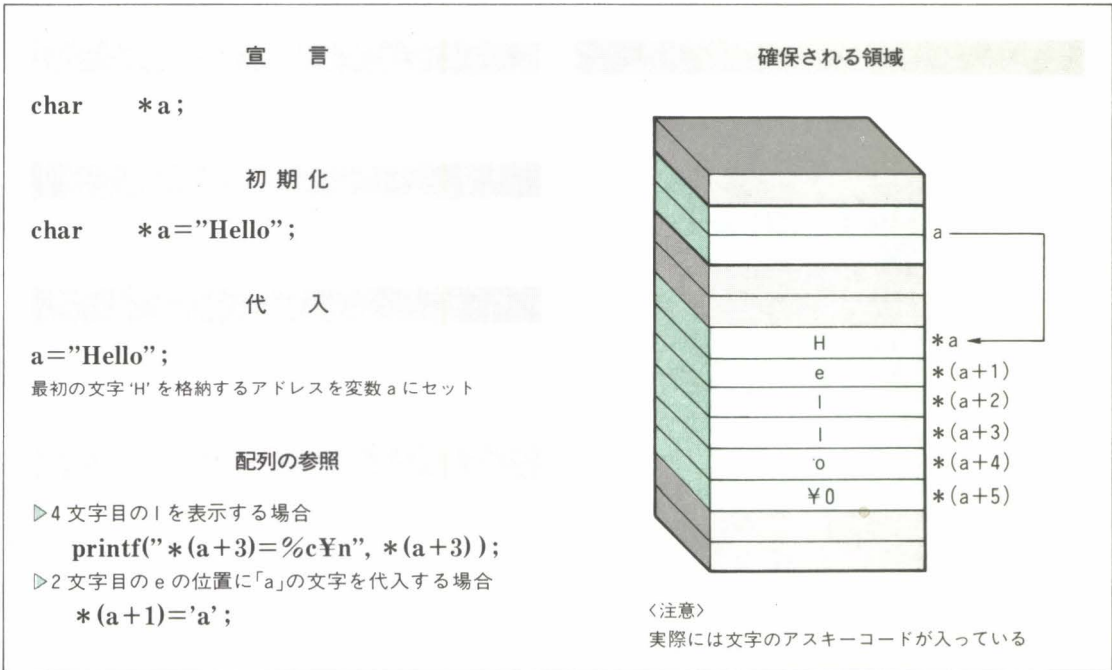



図 5-5　ポインタ変数による文字列の宣言

図 5-5 に示すように，変数 a は 1 文字目の *a('H'が格納されているメモリのアドレス)を指し示しています．ここで，この文字列を表示したい場合には，たとえば printf 関数を使うと，

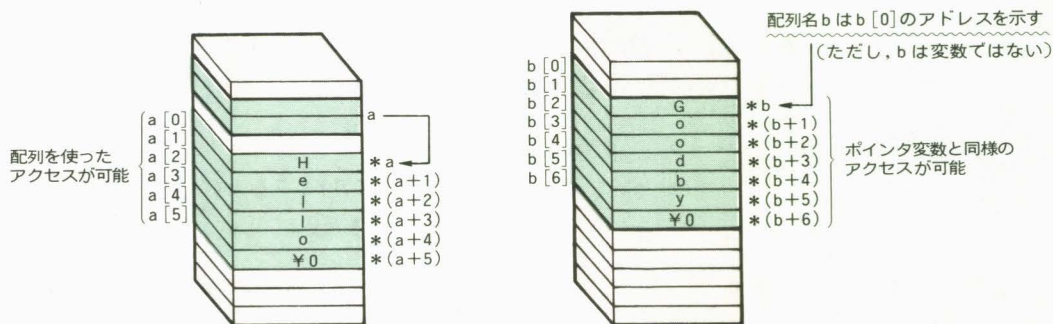
となります．printf 関数は，変数 a で示されるアドレスから順番に 1 文字ずつ読み出しては表示していき，ヌル文字(¥0)があるとそこで表示を終了します．つまり，文字列を管理するためには，先頭の文字が入っているアドレスさえわかればよいことになります(何文字あるかという文字列の長さは問題ではない)．ここで，これまで何度も出てきたヌル文字がC言語の文字列にとっていかに重要かがわか

ります．C言語は，指定されたアドレスからヌル文字までを文字列として認識するのです．これがポインタ変数による文字列の管理方法になります．

配列とポインタ変数は，実は同じように扱うことができます．以下の**リスト 5-2** を見てください．

```
 1: main()
 2:     {
 3:     char *a = "Hello";                 ……ポインタ変数による文字列の定義
 4:     static char b[] = "Goodby";        ……配列による文字列の定義
 5:     int     i;
 6:
 7:     for(i = 0; i < 5; i++)
 8:         printf("a[%d] -----> %c¥n", i, a[i]);……ポインタ変数による文字列は配列
 9:                                                を用いて参照できる
10:     for(i = 0; i < 6; i++)
11:         printf("*(b+%d) ---> %c¥n", i, *(b+i));……配列による文字列はポインタ変数
12:     }                                              を用いて参照できる
```

［実行結果］

```
A>test
a[0] -----> H
a[1] -----> e
a[2] -----> l
a[3] -----> l
a[4] -----> o
*(b+0) ---> G
*(b+1) ---> o
*(b+2) ---> o
*(b+3) ---> d
*(b+4) ---> b
*(b+5) ---> y

A>
```



リスト 5-2　配列とポインタ変数

配列名bがポインタ変数aと同じように，配列の先頭アドレスを表すという点に注目してください．この例からわかるように，配列とポインタ変数は同様に扱えるのです．ただし，このような使い方は読みにくいプログラムの原因にもなりますから，実際のプログラムでは特別に必要がないかぎり〝配列で宣言したものは配列で〃，〝ポインタ変数で宣言したものはポインタ変数で〃使うようにに心がけてください．

文字配列とポインタ変数の扱いについて，**表5-2**に整理しておきます．

| | 文字配列 | ポインタ変数 |
|---|---|---|
| 宣言 | char　a[6]； | char　*a； |
| 初期化 | static　char　a[6] = "Hello"； | char　*a = "Hello"； |
| 値の代入 | a[0] = 'H'；a[1] ='e'；…… | a = "Hello"； |
| データの長さ | 固定 | 初期化，または代入されるまで不定 |
| 各要素の参照 | a[i]　(i = 0, 1, 2 …) | *(a+i)　(i = 0, 1, 2 …) |
| 配列の先頭を指すポインタ | a | a |
| ポインタの演算 | aは定数なので，<br>a = x；, a ++；<br>など値の代入，書き換え不可 | aは変数なので演算可能 |
| 処理速度 | 添字の計算をするため，ポインタ変数より多少遅い | 通常の演算と同じなので，配列の場合より速い |

**表5-2　文字配列とポインタ変数の違い**

文字配列はメモリの配列領域に固定的に確保されますが，ポインタ変数は通常の変数領域に確保され，初期化または代入が行われるまではその領域は不定です．ポインタ変数で宣言した場合も，配列と同じように1文字ずつ代入していくことが可能ですが，その場合は他の変数領域を破壊してしまうこともあるので注意が必要です．

またポインタ変数は，変数ですからこれまで見てきたように演算をすることが可能です．ただし，それは次に挙げる演算に限られます．

- インクリメント，デクリメント　……　a++；，*a--
- 整数との加減算　……　*(a+i)；，a+5；
- ポインタ同士の比較，減算　……　a-b；，a<b；，a! =b；

（注意：aとbが同じ配列を指している場合に限る）

また，配列の場合と同じように，要素数を越えた読み出しや書き込みはチェックされません．

# 5.2　2次元配列とポインタ変数

1次元配列とポインタ変数は，非常に密接な関係にあることが前節でわかったことでしょう．ここでは，2次元配列との関係について話を進めていきます．また最後にその例題として，これまで出てこなかったmain関数の引数についても取り上げることにします．

## ■ 2次元配列

2次元配列では，文字列の配列を扱うことができます．まず，1次元配列と同様に宣言の方法とデータの内部構造を以下の**図5-6**に示しておきます．

2次元配列は「a[0][0]，a[0][1]，a[0][2]，a[1][0]……」という順番でメモリ上に領域を確保します．つまり，1番右の要素から添字が変化していくわけです．また1次元配列と同様に，文字列の最後はヌル文字で終わります．

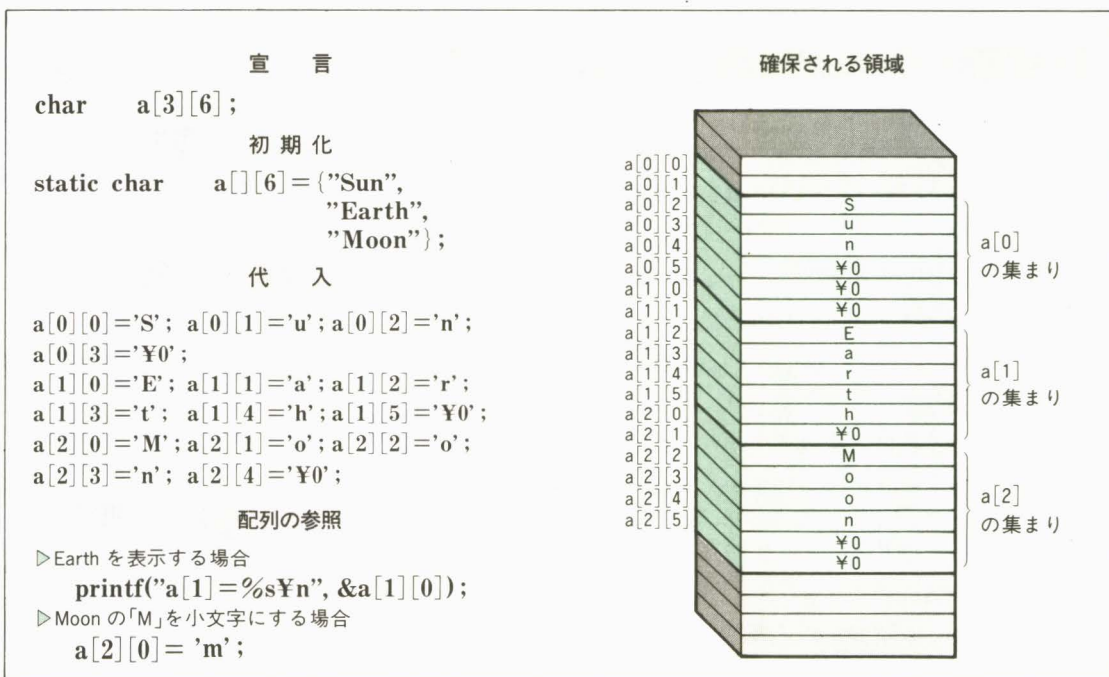



図5-6　2次元配列の初期化とデータの内部構造

配列の要素数は，初期化する場合，行の次元に限って省略することができます(列の次元は省略できません)．また配列の初期化は，次のように1文字ずつ初期化したり，配列の要素ごとに「{}」で囲んで要素のまとまりを明示することも可能です．

```
static char    a[][6] = {'S', 'u', 'n', '¥0', '¥0', '¥0',
                         'E', 'a', 'r', 't', 'h', '¥0',
                         'M', 'o', 'o', 'n', '¥0', '¥0'};
static char    a[][6] = {{'S', 'u', 'n'},
                         {'E', 'a', 'r', 't', 'h'},
                         {'M', 'o', 'o', 'n'}};
```

## ■ ポインタ変数の配列

ポインタ変数は文字列を扱えますから，その配列を作ることで2次元配列と同様に「文字列の配列」を扱うことが可能です．以下の図5-7に，その宣言と初期化，およびデータの内部構造についてまとめます．

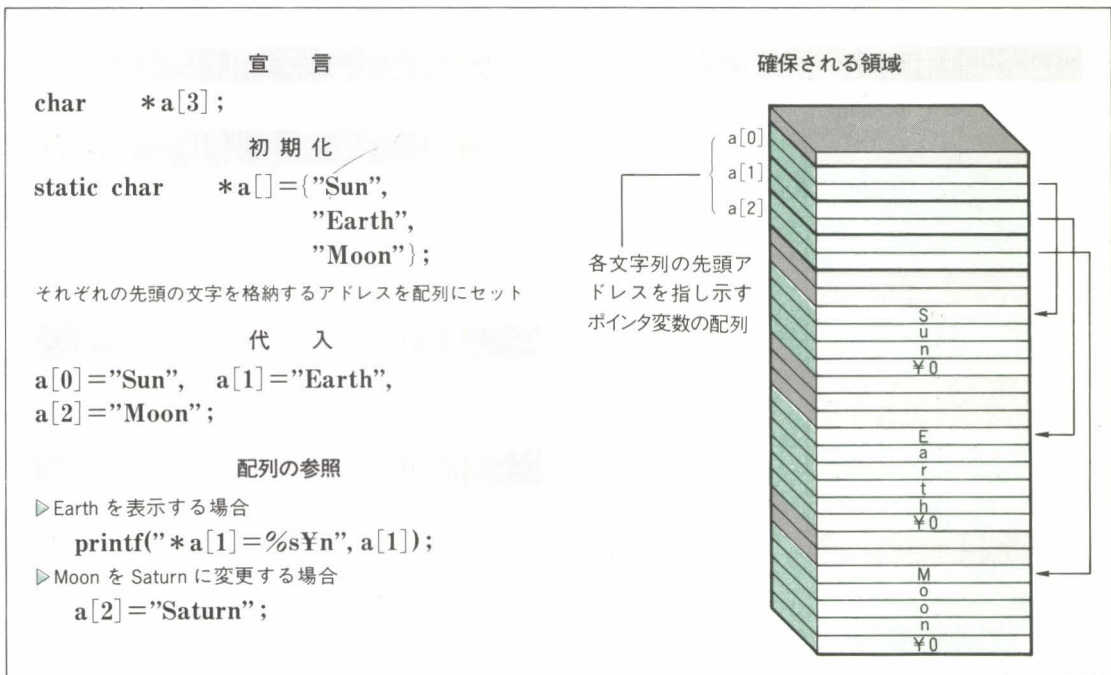
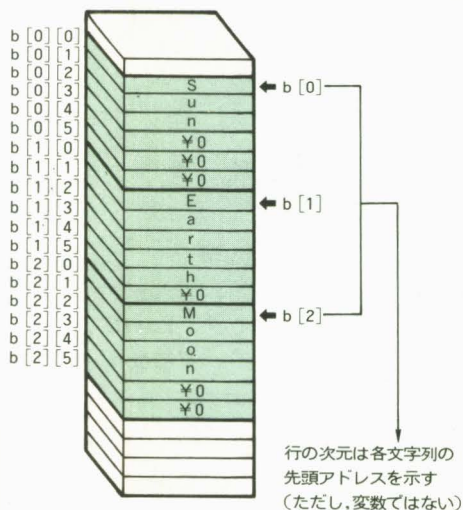
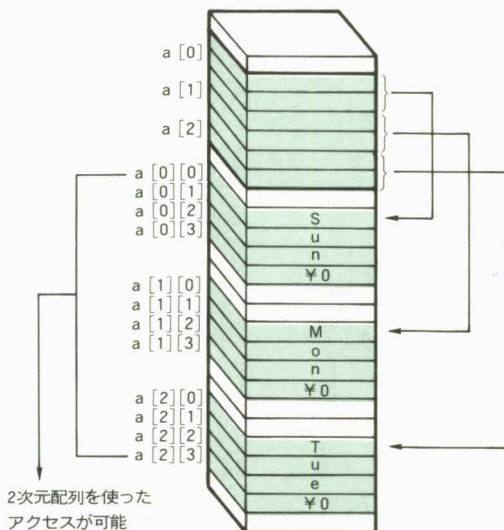



図5-7　ポインタ変数の配列

```
 1: main()
 2:     {
 3:     static char   *a[] = {"Sun", "Mon", "Tue"}; ………ポインタ変数の配列を使った文字列の表現
 4:     static char   b[3][6] = {"Sun", "Earth", "Moon"}; ………2次元配列を使った文字列の表現
 5:     int     i, j;
 6:
 7:     for(i = 0 ; i < 3 ; i++) ………ポインタ変数の配列を2次元配列としてアクセスする
 8:      for(j = 0 ; j < 3 ; j++)
 9:            printf("a[%d][%d] -----> %c¥n", i, j, a[i][j]);
10:
11:     for(i = 0 ; i < 3 ; i++) ………2次元配列をポインタ変数としてアクセスする
12:      printf("b[%d]     -----> %s¥n", i, b[i]);
13:     }
```

［実行結果］

```
A>test
a[0][0] -----> S
a[0][1] -----> u
a[0][2] -----> n
a[1][0] -----> M
a[1][1] -----> o
a[1][2] -----> n
a[2][0] -----> T
a[2][1] -----> u
a[2][2] -----> e
b[0]    -----> Sun
b[1]    -----> Earth
b[2]    -----> Moon

A>
```



リスト5-3　2次元配列とポインタ変数の配列

**図 5-7** を見てわかるように，配列の要素には各文字列の先頭のアドレスが入ります．そしてC言語の文字列はかならずヌル文字で終わりますから，各文字の先頭アドレスを指し示すポインタ変数の配列を作れば，文字列の配列は管理できます．

さて，先と同様に2次元配列とポインタ変数の配列を比較してみましょう．**リスト 5-3** を見てください．

ここで2次元配列の配列名(a[0]～a[1])は，ポインタ変数と同じように文字列の先頭のアドレスを指しています．また逆に，ポインタ変数の配列も2次元配列を使って同じように1文字ずつアクセスすることができます．

以下の**表 5-3** に2次元配列とポインタ変数の配列について整理しておきます．

| | 2次元配列 | ポインタ変数の配列 |
|---|---|---|
| 宣言 | char　a[2][4]; | char　*a[]; |
| 初期化 | static　char　a[][4] = {"Sun", "Mon"}; | static　char　*a[] = {"Sun", "Mon"}; |
| 値の代入 | a[0][0] = 'S'; a[0][1] = 'u'; …… | a[0] = "Sun"; …… |
| 各要素の参照 | a[i][j]　(i = 0, 1 … ／ j = 0, 1 …) | *(a[i]+i)　(i = 0, 1 … ／ j = 0, 1 …) |
| 配列の先頭を指すポインタ | a[i]　(i = 0, 1, 2,…) | a[i]　(i = 0, 1, 2 …) |
| 確保される領域 | 固定<br>2×4のメモリが割り当てられる<br>S　a[0][0]<br>u　a[0][1]<br>n　a[0][2]<br>¥0　a[0][3]<br>M　a[1][0]<br>o　a[1][1]<br>n　a[1][2]<br>¥0　a[1][3] | 不定<br>2個のポインタを格納する領域と，初期値または代入に応じてメモリが確保される<br>a[0]<br>a[1]<br>*a[0]　S<br>*(a[0]+1)　u<br>*(a[0]+2)　n<br>*(a[0]+3)　¥0<br>*a[1]　M<br>*(a[1]+1)　o<br>*(a[1]+2)　n<br>*(a[1]+3)　¥0 |

**表 5-3　2次元配列とポインタ変数の配列**

ポインタ変数の配列では，配列自身はメモリの配列領域に確保されますが，その実体である文字列は通常の変数領域に確保されます．またその変数領域は，初期化または代入が行われるまで不定になります．

ここで大事なことは，ポインタ変数の配列を使うと文字列の長さを気にする必要がないという点で

す．２次元配列では固定的にその領域が確保されるので，扱う文字列の最大の長さを考慮しなければなりません．つまり，任意長で長さの異なる文字列を扱う場合には，「ポインタ変数の配列」が圧倒的に有利です．

## ■ ポインタ変数のポインタ変数

ポインタ変数はこれまでに示した使い方のほかに，ポインタ変数自身のポインタ変数を使うこともできます．つまり，「ポインタ変数が存在するアドレスを指し示すポインタ変数」を宣言することができるのです．たとえば，

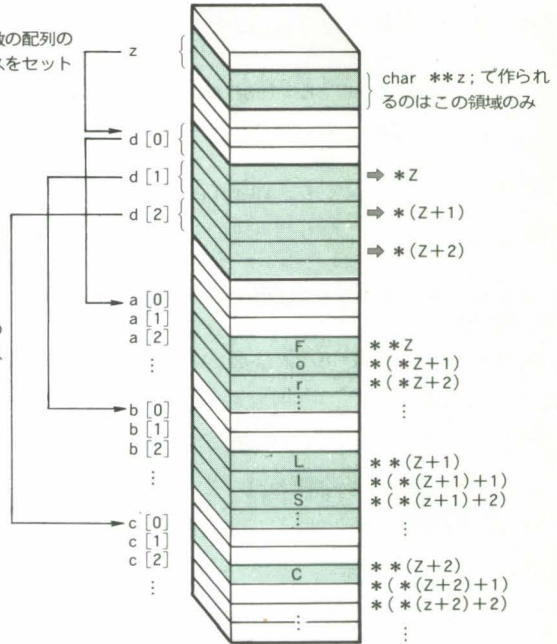
```
char    **a;
```

と宣言された場合は，「char　*a;」のときと同様に，

**a …… char 型の変数
*a ……「**a」のある位置を指し示すポインタ変数
a ……「*a」というポインタ変数のある位置を指し示すポインタ変数

という3つの変数が宣言されたと考えることができます．ただし，ここで宣言されるのは，ポインタ変数の器だけなので配列の場合と違いその実体(**a として参照される文字変数そのもの)がありません．つまり，直接この変数を初期化したり，この変数に対して代入することはできないのです．それでは，この変数はいったいどのように使われるのでしょうか？　まず，その利用例を以下の**リスト5-4**に示すことにします．

```
 1: char    a[] = "Fortran";    各文字列の初期化
 2: char    b[] = "LISP";
 3: char    c[] = "C";
 4: main()
 5:     {
 6:     char    *d[4]; ………ポインタの配列を宣言
 7:     d[Ø] = a;     d[1] = b;    関数 a[ ]のポインタ
 8:     d[2] = c;     d[3] = Ø;    各配列に関数のポインタを渡す　終わりの印
 9:     display(d); ………ポインタ配列のポインタを引数として渡す
1Ø:     }
11: display(z)
12:     char    **z; ………ポインタ変数のポインタ変数として引数を宣言
13:     {
14:     int     i;
15:     for(i = Ø ; *(z + i) != Ø ; i++)
16:         printf("Language %d is %s¥n", i, *(z + i));
17:     }
```

ポインタ変数のポインタ変数をインクリメントしていくことで，各配列の文字列が表示できる

リスト 5-4　「ポインタ変数のポインタ変数」の利用例

この例でわかるように，長さの異なる文字列を引数として別の関数に渡したい場合などに，ポインタ変数のポインタ変数を使うとその取り扱いが楽になります．

ここで，「ポインタ変数のポインタ変数」は，実は先の「ポインタ変数の配列」とまったく同じものです．つまり，「char　*a[20]」という宣言は以下のように分けて考えることができます．

|  |  |  |
|---|---|---|
| *a[n] | …… | char 型の変数の1つ |
| a[n] | …… | char 型の変数のポインタ変数の配列 |
| a | …… | char 型のポインタ変数の配列のアドレスを指し示すポインタ変数 |

## ■ main 関数の引数

main 関数の引数はこれらのポインタ変数の例題として，ちょうどよいサンプルです．また，これから数多く使う機会があると思いますので，ここでくわしく取り上げます．

C 言語のプログラムは main という名前の関数から始まりますが，この関数も他のライブラリ関数や，自作の関数のように「引数」を持つことができます．

たとえば，以下のMS-DOSのcopyコマンドやCP/Mのpipコマンドのように，コマンドの実行と同時に値や文字列を取り込みたい場合があるでしょう．

A>copy　a：test1.c　b：test2.c
A>pip　b：test2.c ＝ a：test1.c

自作のプログラムでも，main関数の引数を使うことで次のようにコマンドライン上の文字列を使用することが可能になります．

A>test　－s　prog1.c　prog2.c

実際のmain関数の引数は2つあります（本当はもう1つあるのですが，ここではこの2つに限っておきます）．慣用的にこの引数の名前は，それぞれargc（Argument-Count）とargv（Argument-Value）となっていますが，もちろんどんな名前でも構いません．また，これらの引数の名前を省略して，「ac」，「av」というのも多く使われています．

まず，コマンド行に打ち込まれた文字列をすべて表示するプログラムを書いてみましょう（**リスト5-5**）．

**リスト5-5**からわかるように，最初の引数はint型で，次の引数が「ポインタ変数の文字配列」になっ

```
 1: main(argc, argv)
 2:     int argc;  ……………コマンドライン上の引数の数(コマンド名も含む)がOSから渡される
 3:     char *argv[];  ………引数が格納されているメモリ上のアドレスが配列としてOSから渡される
 4:     {
 5:     int i;
 6:
 7:     printf("argc = %2d¥n¥n", argc);
 8:
 9:     for(i = 0 ; i < argc ; ++i)
10:         printf("Argument No.%2d  : %s¥n", i, argv[i]);
11:     }                                          └ 文字列の先頭アドレスを渡す
```

［実行結果］

```
A>test -s prog1.c prog2.c↵
argc =  4

Argument No. 0  : test  ………MS-DOS Ver.2.XX，CP/Mではコマンド名が入らずヌル文字となる
Argument No. 1  : -s
Argument No. 2  : prog1.c
Argument No. 3  : prog2.c

A>
```

リスト5-5　コマンドライン上の文字列を表示するプログラム（1）

ています．これら2つの変数にコマンドの引数の情報がすべて詰まっています．

まず「argc」には，コマンド行の空白文字で区切られた文字列の数が渡されます．ここで，コマンド名そのものも数に含まれていることに注意してください．

「argv」には，起動時に打ち込まれた各文字列のアドレスがオペレーティング・システムから渡されます．**リスト 5-5** の例では，文字列「test」(コマンド名)が格納されている配列は，「argv[0]」を先頭アドレスとする位置に存在します．以下の引数もまったく同じように，ポインタ変数の配列に文字列の先頭アドレスが格納されています．**図 5-8** にこのときのメモリ上のレイアウトを示しておきましょう．

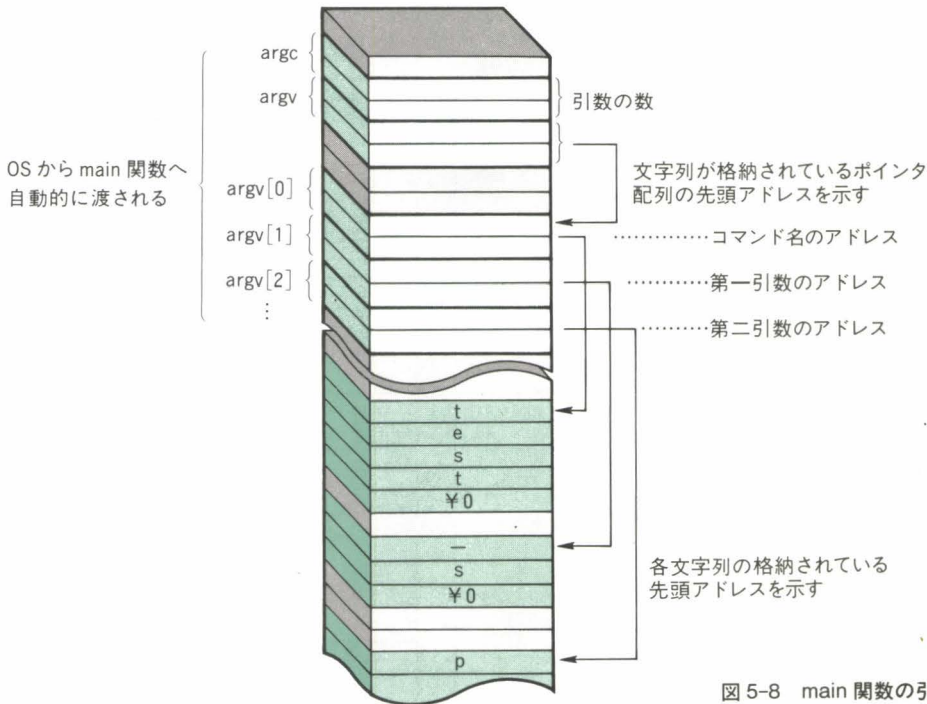



**図 5-8　main 関数の引数**

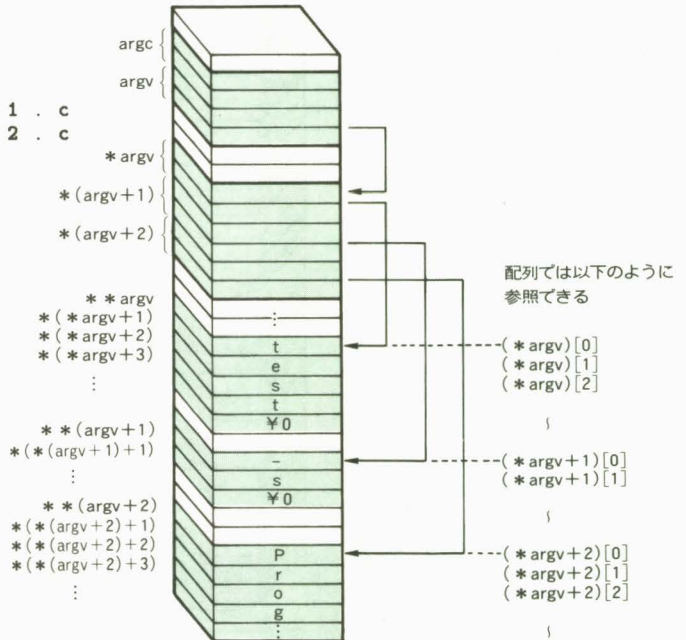
コマンドラインから渡される引数は，すべて「Cストリング」に整形されていますから，文字列としてそのまま使うことができます．もし，整数や浮動小数点数として使用したい場合は，「atoi」や「atol」などの関数を用いて必要なデータ型に変換してください．

また，引数のアドレスが渡される「argv」は「ポインタ変数のポインタ変数」として宣言することもできます．先の**リスト 5-5** を「** argv」で宣言し，今度は引数を1文字ずつ取り出すように変更してみましょう(**リスト 5-6**)．

```
 1: main(argc, argv)
 2:     int argc;
 3:     char **argv;
 4:     {
 5:     int i,j;
 6:
 7:     for(i = 0 ; i < argc ; i++, argv++)
 8:         {
 9:         printf("No.%2d ---> %s : ", i, *argv);
10:
11:         for(j = 0 ; *(*argv+j) ; j++)
12:             printf(" %c", *(*argv+j));
13:
14:         printf("¥n");
15:         }
16:     }
```

3行目：ポインタ変数のポインタ変数としてargvを宣言
7行目 argv++：次の文字列へ
9行目 *argv：引数として渡された文字列の先頭アドレス
12行目 *(*argv+j)：引数として渡された1文字．argv自身は7行目でインクリメントされる点に注意
配列を使うと「(＊argv)[j]」と書くこともできる

［実行結果］

```
A>test -s prog1.c prog2.c
No. 0 ---> test :  t e s t
No. 1 ---> -s :  - s
No. 2 ---> prog1.c :  p r o g 1 . c
No. 3 ---> prog2.c :  p r o g 2 . c

A>
```



**リスト5-6　コマンドライン上の文字列を表示するプログラム(2)**

ここでポインタ変数が2つ重なったときには，その優先順位に気をつけてください．「＊(＊argv＋1)」と「＊＊argv＋1」ではその内容がかなり異なってきます．ポインタ変数は確かにその扱いは便利なのですが，それがいくつも重なってしまうとその実体が何であるかわかりにくくなってしまいます．そのようなときには，**リスト5-6**に示すようにメモリ上の図を書いてみるとよいでしょう．

## ■ポインタ変数と優先順位

ポインタ変数を使っているとその優先順位が問題となることがあります．ここではまとめとして，ポインタ変数についていくつか注意事項を述べておきます．

まず，以下の2つの宣言の違いをみてみましょう．

① char　(*b)[6]；　……　6つの領域を持ったchar型の配列へのポインタ
　　　　　　　　　　　　(char　b[][6]；と同じ)
② char　*b[6]；　……　char型のポインタが6つある配列

ポインタを示す単項演算子「*」と配列を表す「[]」では，配列の方が優先順位が高くなります(43ページの演算子の優先順位表を参照)． ここで， 宣言を読むときには， 逆に優先順位の低いものから読んでいくとよいでしょう．つまり，②では配列の方が優先順位が高いので「ポインタの配列」となり，①ではそれが逆になるので「配列へのポインタ」となるわけです．実際にこの両者の違いは，以下の**図5-9**に示した宣言時に確保される領域を見ればよくわかります．

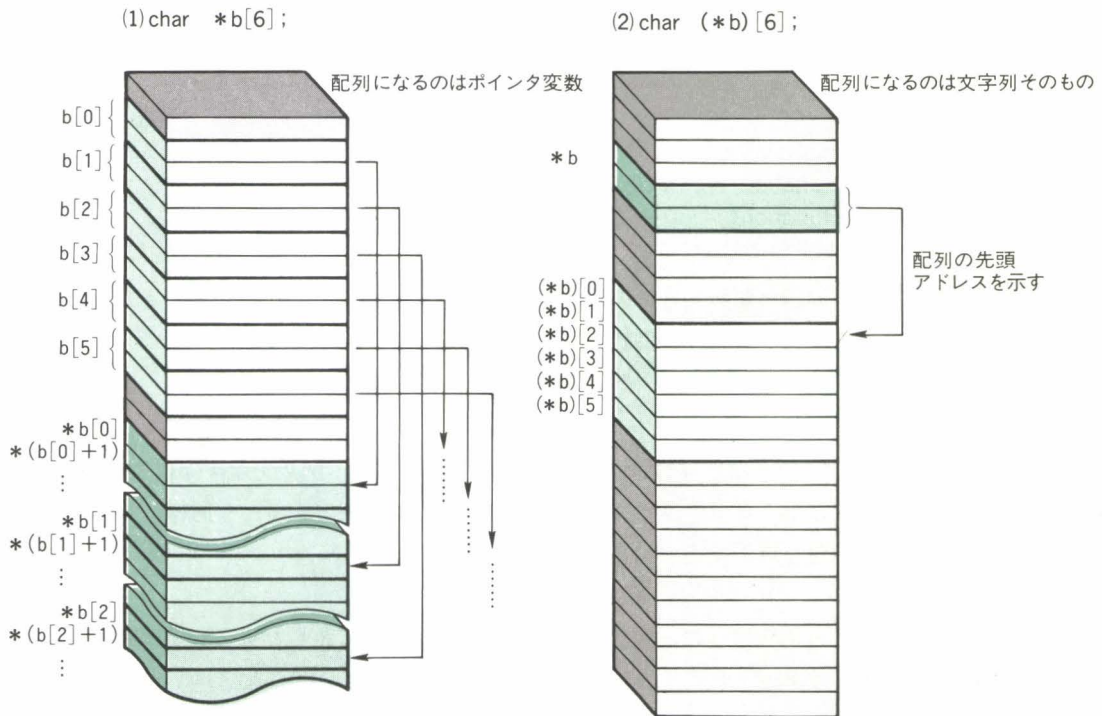



図5-9　「char　(*b)[6]；」と「char　*b[6]；」の違い

# 5.3　関数とポインタ

ポインタ演算子は，変数だけでなく関数にも使うことができます．つまり，「その関数が存在するアドレス」をプログラム中で利用することが可能です．

システム標準関数なかには，たとえばsort関数のように引数として「関数のポインタを渡す」という仕様のものがいくつかあります．そこでここでは，関数へのポインタについて解説した後，クイックソートを行う「qsort」関数を使った例題を示してみることにします．

なお，関数の書式や使い方については，「第6章 関数」で解説しています．

## ■ 関数へのポインタ

次のような関数の使用宣言を考えてみましょう．

```
int  function()；  …… int型の返値を持つ関数functionの宣言
```

この宣言では，ポインタ変数の場合と同じように，2つの実体が定義されたと考えることができます．

| | |
|---|---|
| function | …… 関数functionが存在するアドレス |
| function() | …… 関数functionそのもの |

つまり，配列の宣言と同様に関数名はその関数が存在するアドレスを指し示しています．ここで，ポインタ演算子を使うと，以下のように関数を呼び出すこともできます．

```
(*function)()；  …… functionが指し示す関数の実体を実行する
```

多くの場合「関数のポインタ」は，「関数の引数」―― つまり，関数へ関数を渡す際に使われます(関数の引数についての解説は第6章で行う)．これは，関数が変数や配列とまったく同様に扱えることを意味しています．その例題を以下に1つ示してみましょう(**リスト5-7**)．

次ページの**リスト5-7**にあるように，関数のアドレスを引数として渡した場合，受け側の関数ではかならず「関数へのポインタ」として宣言します．

```
 1: main()
 2:      {
 3:      int     a, b, function();        ── int 型の返値を持つ関数 function の使用宣言
 4:
 5:      a = 3;     b = 2;
 6:                                       ── 関数のアドレスを渡す
 7:      printf("%d + %d = %d¥n", a, b, execute(function, a, b));
 8:      }
 9:
10: execute(func, x, y) ………関数を実行するだけの関数
11:      int     (*func)(), x, y; ………関数 func へのポインタを使用するという宣言
12:      {
13:      return((*func)(x, y)); ………関数 func の実体を呼び出して実行し，main 関数へ戻る
14:      }
15:
16: function(p, q) ………execute 関数の仮引数 func で宣言しないこと
17:      int     p, q;
18:      {
19:      return(p + q);
20:      }
```

［実行結果］

```
A>test
3 + 2 = 5

A>
```

リスト 5-7　関数へのポインタの使用例

## ■ 関数へのポインタの配列

関数へのポインタの配列を使うと，多くの関数を同じような形で呼び出すことが可能になります．先の**リスト 5-7** を，4つの関数を呼び出すように変更してみましょう(**リスト 5-8**)．

このリストでわかるように，各関数のアドレスをそれぞれ配列にセットしておき，その配列の要素を execute 関数に渡すことで次々に関数を実行していきます．

なお，関数へのポインタを使う場合，その優先順位に注意してください．以下にその例をいくつか挙げておきます．この場合も優先順位の低いものから読んでいくと，その構造と一致します．

```
int      (*function)()    …… int 型の関数へのポインタ
int       *function()     …… int 型のポインタを返す関数
int    (*function[])()    …… int 型の関数へのポインタ配列
```

```
 1: main()
 2:     {
 3:     int     a, b, i;
 4:     int     func1(), func2(), func3(), func4(),(*function[4])();
 5:
 6:     function[0] = func1;     function[1] = func2;
 7:     function[2] = func3;     function[3] = func4;
 8:
 9:     a = 3;     b = 2;
10:
11:     for(i = 0 ; i < 4 ; i++)
12:         printf("func No.%d -----> %d¥n", i+1, execute(function[i], a, b));
13:     }
14:
15: execute(func, x, y)
16:     int     (*func)(), x, y;
17:     { return((*func)(x, y)); }
18:
19: func1(p, q)
20:     int     p, q;
21:     { return(p + q); }
22:
23: func2(p, q)
24:     int     p, q;
25:     { return(p - q); }
26:
27: func3(p, q)
28:     int     p, q;
29:     { return(p * q); }
30:
31: func4(p, q)
32:     int     p, q;
33:     { return(p / q); }
```

関数へのポインタの配列を宣言

配列の各要素の関数のアドレスをセット

関数のアドレスを順番に execute 関数に渡す

[実行結果]

```
A>test
func No.1 -----> 5
func No.2 -----> 1
func No.3 -----> 6
func No.4 -----> 1

A>
```

リスト 5-8　関数へのポインタの配列の使用例

## ■ qsort 関数の利用例

ここでは「関数へのポインタ」の応用として，qsort(クイックソート)関数の利用例を取り上げます．まず，関数の仕様を**表 5-4** に示します．

| 関数名 | qsort | |
|---|---|---|
| ヘッダファイル | search.h | |
| 返値 | void（なし） | |
| 書式 | qsort (base, num, width, compare) ;<br>　char　　＊base ;<br>　unsigned (char/short/int/long) num ;<br>　unsigned (char/short/int/long) width ;<br>　int　　(＊compare) ; | |
| 引数 | base | ソートする要素の配列の先頭アドレス |
| | num | ソートする要素の数 |
| | width | ソートする要素の配列のバイト数 |
| | compare | ソートの比較に使われる関数へのポインタ |

〈注意〉
• 必要なヘッダファイルや書式は異なる処理系がある．APPENDIX を参照のこと

**表 5-4　qsort 関数の仕様**

qsort 関数を使う場合，ソートの比較に使われる関数の実体(＊compare())は利用するユーザーが作成しなければなりません．それは以下のような仕様を満たしている必要があります(**表 5-5**)．

| 引数 1 | 比較される配列要素のポインタ |
|---|---|
| 引数 2 | 比較されるもう 1 つの配列要素のポインタ |
| 返値 | 比較の結果．引数 1 ＜ 引数 2→(－1)<br>引数 1 ＝ 引数 2→0<br>引数 1 ＞ 引数 2→1 |

**表 5-5　ユーザーが作成する関数の仕様**

それでは実際に，qsort 関数を使ったプログラムを組んでみることにしましょう．このプログラムはキーボードから文字列を読み込み，その文字列の長さの順に並べ替えを行います(**リスト 5-9**)．

```
1: #include        <stdio.h>
2: #include        <search.h>  ………qsort 関数が定義されているヘッダファイル
3: #include        <ctype.h>   ………strlen 関数が定義されているヘッダファイル
4:
5: #define         MAXDATAS        20  …………ソートされる文字配列の要素の最大数
6: #define         MAXCHARS        20  …………文字配列の 1 要素の最大文字数
7:
```

```
 8: int      lencmp(ax, ay) ………2つの文字配列の長さを比較して結果を返す関数
 9:     char     **ax;  } 引数は文字列が格納されたアドレスを持つポインタ変数
10:     char     **ay;  }
11:     {
12:     if(strlen(*ax) < strlen(*ay))    return(-1); ………第2引数の方が長い場合
13:     if(strlen(*ax) > strlen(*ay))    return(1);  ………第1引数の方が長い場合
14:                                      return(0);  ………第1引数と第2引数が同じ長さの場合
15:     }
16:
17:
18: main()
19:     {
20:     int      i, j;
21:     char     *point_data[MAXDATAS]; ……………入力データ格納用のポインタ配列
22:     char     buff[MAXDATAS][MAXCHARS]; ………ダミー用の2次元配列
23:
24:     for(i = 0 ; i < MAXDATAS ; ++i)
25:         {
26:         printf(" Input strings [%02d] : ", i+1);
27:         point_data[i] = &buff[i][0]; ………配列の先頭アドレスを明確に渡す
28:         gets(point_data[i]); ……………………文字列の取り込み
29:         if(0 == *point_data[i]) break;
30:         }
31:                                                 ┌関数のポインタを渡す
32:     qsort(point_data, i, sizeof(char *), lencmp); ………ソート関数へデータを渡す
33:
34:     printf("¥n<Sorting>¥n"); ………ソートされたデータの表示
35:     for(j = 0 ; j < i ; ++j)
36:         printf(" %2d: %s¥n", j+1, point_data[j]);
37:
38:     }
```

［実行結果］

```
A>test↵
 Input strings [01] : January↵
 Input strings [02] : February↵
 Input strings [03] : March↵
 Input strings [04] : April↵
 Input strings [05] : May↵
 Input strings [06] : June↵
 Input strings [07] : ↵

<Sorting>
  1: May
  2: June
  3: March
  4: April
  5: January
  6: February

A>
```

リスト5-9　qsort 関数の利用例

qsort 関数では，ソートの対象となる文字列や数値を直接扱うことはできません．main 関数の引数の argv のように「ポインタ変数のポインタ変数」として扱います．以下の**図 5-10** に，qsort 関数に渡す引数の扱いについてまとめておきます．

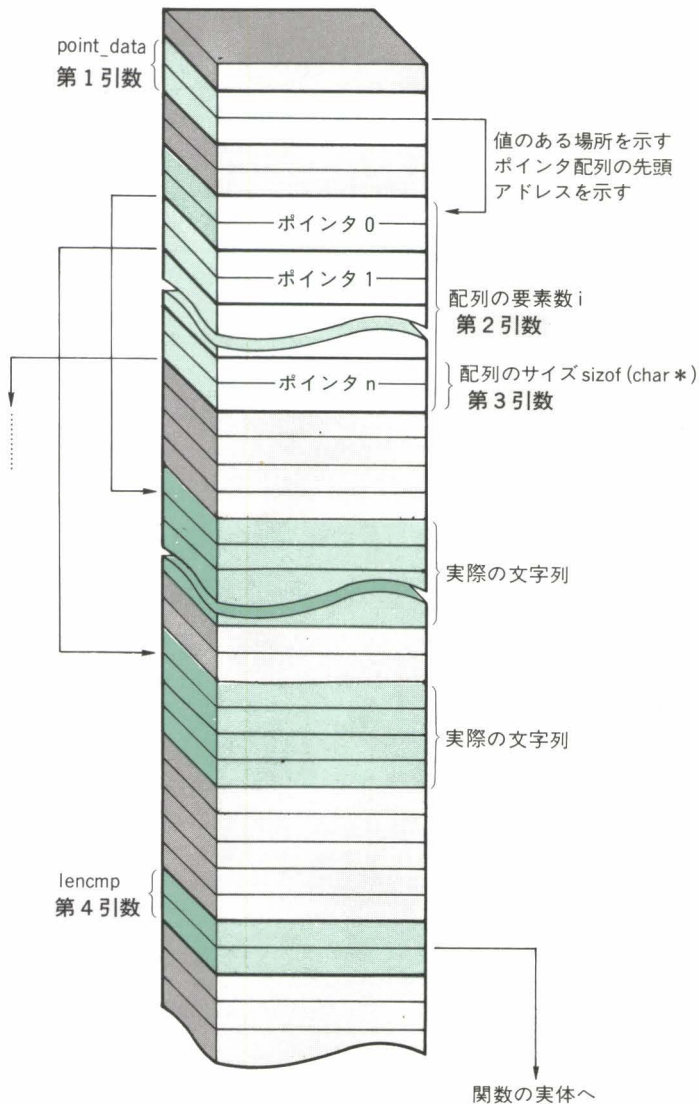
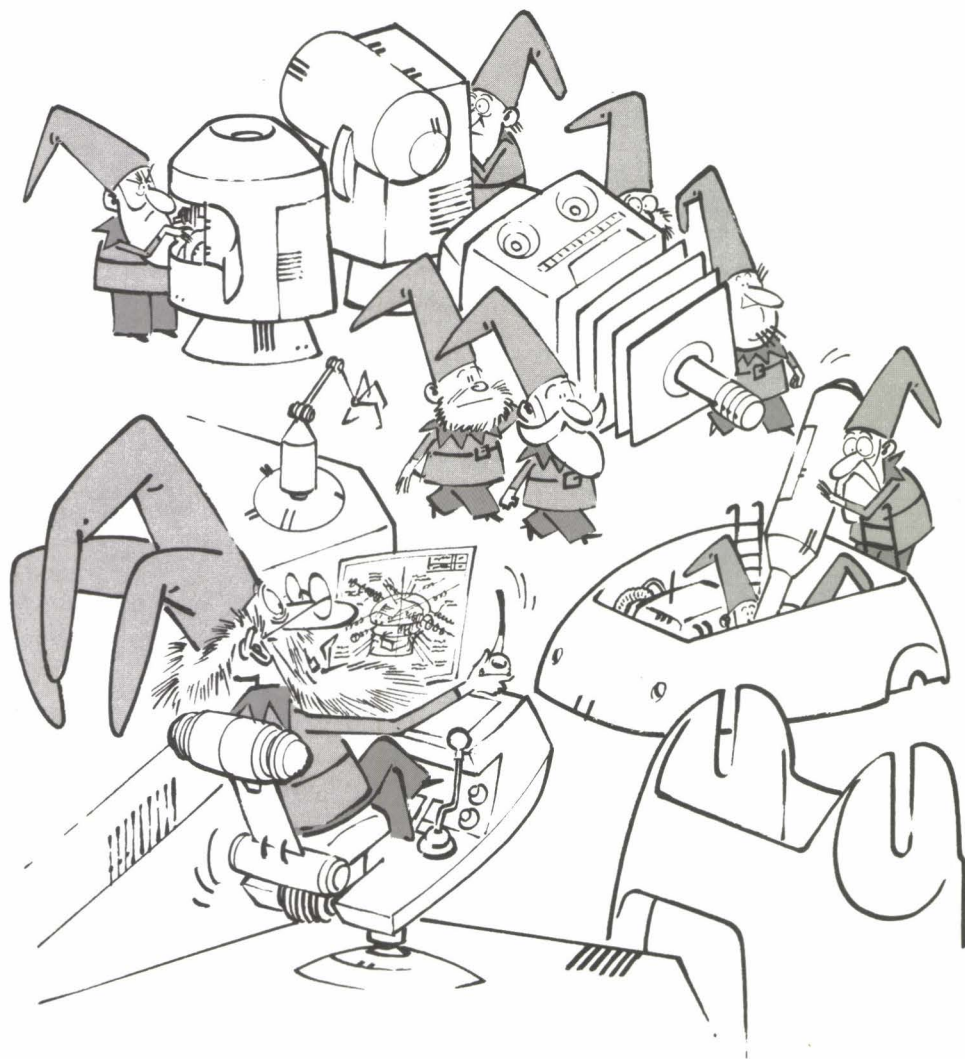



図 5-10　qsort 関数の引数の渡し方

# 第6章
# 関　数

C言語のプログラムは，関数の集まりとして構成され，関数1つ1つが大きな実行の単位となっています．C言語の関数は，他の言語でいうサブルーチンとは大きく異なります．それはC言語の関数が，数学で使われる関数と同じように「引数」と「返値(関数値)」を持っているという点にあります．これによりC言語では，非常にスマートなプログラムが書きやすい構造になっています．最初はこの関数の考え方を理解するのにとまどうことがあるかもしれませんが，慣れてしまえば実にスッキリとプログラムを記述できます．

この章では，こういった関数の構造と使用法について解説していくことにしましょう．

# 6.1　関数の書式と使い方

関数を利用するには，C言語コンパイラにあらかじめ付属しているシステム標準関数を使う場合と自分で作成した関数を使う場合の2通りがあります．いずれにしても，関数の書式やその返値の扱いについて知っておかなければなりません．この節では，このような関数を扱う上で基礎となる事項についてまとめておきます．

## ■関数とは

C言語で使われる関数は，その呼び出し方が数学で使われる関数とまったく同じです．たとえば，次の例を見てみましょう．

a ＝ function (x, y)；　…… 変数x, yを関数functionに入力し，その返値(関数値)を変数aに代入する

つまり関数は，「ある一定の入力に対して，一意に結果を出力するもの」と定義できます．ただし，数学での関数は変数x，yによってのみ変数aの値が決まるわけですが，C言語の関数の中身は実際にはプログラムですからxとyだけが出力結果に影響を与えるわけではありません．

関数は，その内部でどのような処理が行われているのかを知らなくとも，入力する値と出力される結果が何であるかがわかっていれば使うことができます．また関数自身は，入力と出力に影響を及ぼさない限り，内部の仕様は自由に設定／変更することが可能です．

このような関数を**モジュール**と呼びます．つまり，C言語では，互いに影響を与えない独立したモジュールを組み合わせることでプログラムを作成していきます．実際のプログラミングでは，各モジュールごとに完成させていけばよいので，プログラミングとデバッグを非常に効率よく行うことができます．

## ■関数の定義

C言語の関数は，以下のように定義できます．

```
記憶クラス　データ型　関数名(仮引数のリスト)
    仮引数の宣言；
    {
    関数の本体(実行文)
    }
```

これを一般の変数を宣言する場合と比べてみましょう．変数では以下のようになります．

**記憶クラス　データ型　変数名；**

つまり変数も関数も「数」ですから，それを使う(呼ぶ)側にとっては，まったく同じものとして扱えるわけです．

C言語の関数には，引数のない関数，返値(関数値)を持たない関数(Pascalでいう手続き)も存在しますが，ここでは上記に示した関数の定義にそって各事項の解説を行います．

## ■関数の記憶クラス

関数の記憶クラスは，「static」と「extern」のみが指定できます(「4.4 記憶クラス」を参照)．まず，この違いを以下の**表6-1**に示しておきます．

| 記憶クラス | 機能 |
|---|---|
| static | 宣言されたコンパイル単位でのみ使用可能 |
| extern | 宣言されたコンパイル単位以外でも使用可能(外部関数) |

〈注意〉
• 通常，関数の記憶クラスは省略されるが，その場合は「extern」として扱われる．

**表6-1　関数の記憶クラス**

関数の記憶クラスを意識しなければならないのは，プログラムをいくつかのソースファイルに分割してコンパイルする場合です(「9.2 分割コンパイル」を参照)．「static」と「extern」で宣言した場合の関数の呼び出し関係を以下の**図6-1**に示します．

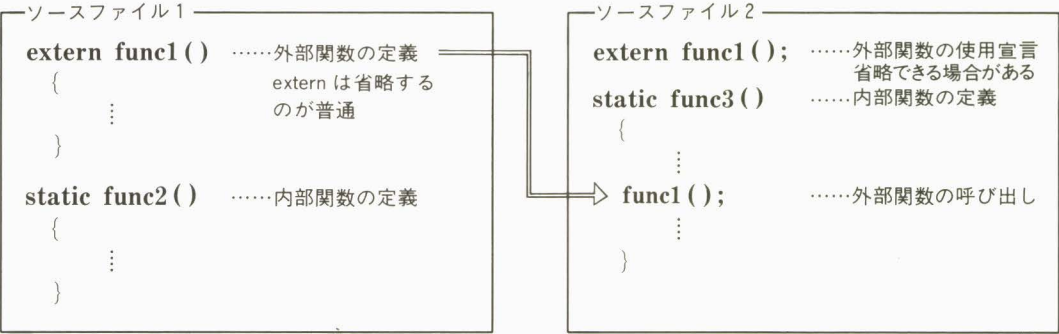



図6-1　static と extern で宣言した場合の関数の呼び出し関係

この図でわかるように，staticで関数を宣言した場合は，そのソースファイルに特有の関数となります．たとえば，同名の関数を別のコンパイル単位で使用するときにはstaticで宣言しなくてはなりません．

## ■関数の返値とデータ型

関数で返値(関数値)を返す場合は，関数本体のなかで以下のようにreturn文を用いて行います．

```
return(式)；
```

ここで式の結果返される値の型が何であるかを宣言しておかなければなりません．それが前述の関数の定義で宣言されるデータ型です．このデータ型は，「4.3 データ型」で示したすべてのデータ型を使用することができます（77ページの**表4-6**を参照）．また，各データ型のポインタ変数を返値として返すことも可能です．つまり，C言語で扱える「値」はすべてこの関数値として使うことができます．

また，これ以外に変数には用意されていない，関数特有のデータ型もあります．これは「void型」と呼ばれます．

void　…… 返値がないことを明示

void型は，C言語の処理系によっては使えないものもありますが，最近の処理系では多くのものが採用しているようです(とくに，ANSI規格に準拠した処理系)．voidは関数の宣言時に用いるものですが，voidのない処理系では「返ってくる値があっても無視する」という使い方で代用することもできます．

返値は扱える値が1つに限られているので，呼んだ先の関数でエラーの有無を確認する場合や簡単な動作の関数で値を渡す場合の入れ物として使うことが多いようです．たとえば，システム標準関数でファイルのオープンに使われるfopen関数については，**表6-2**のように返値があらかじめ決められています．

| **書式** | FILE　fopen (filename, mode)； |
|---|---|
| **返値の意味** | 0　　：ファイルのオープンに失敗した<br>0以外：ファイルのオープンに成功した<br>値はファイル構造体へのポインタを示す |

**表6-2　fopen関数の返値**

## ■関数の使用宣言

関数は変数と同様に，使用する前にあらかじめ使用宣言をしておかなければなりません．その書式を以下に示します．

| 記憶クラス　データ型　関数名()； |
|---|

記憶クラスとデータ型については，前述の説明の通りです．また，実際の関数で仮引数を使用する場合でも，仮引数は省略して宣言します．

ただし，以下の場合は関数の使用宣言を省略することができます．

① 関数の返値が int 型である場合
② 使用する前に関数の定義が行われている場合

コンパイラにとっては「宣言や定義をした後で呼び出す」ほうが自然なのですが，C 言語では「呼ぶ関数が後にあると仮定して呼び出しを行う」という書き方が一般的になってしまったため，例外的に「まだ宣言していない関数でも int 型ならば呼ぶことができる」というような仕様を認めているのです．しかし int 型以外のデータ型を返す場合は，前もって使用宣言をするかその定義をしておかなければなりません．

以下の図 6-2 にその例を示します．

```
extern long    sub(); ………外部関数の使用宣言
                        関数の外で宣言しているので，どの関数でも使用可能
void func1() ………void 型は返値を返さないことを明示する
    {
    long    a;
        :
    a = sub(); ………外部関数の呼び出し
        :
    }

float func2()
    {
    int     b;
        :
    b = func3(x, y); ………関数 func3 は前に宣言または定義がないので，
                          int 型の返値を持つと仮定して呼び出される
        :
    return((float)b);
        :
    }
```

```
int func3(p, q) ………int は省略可能
    int    p, q;
    {
           :
    func1(); ………関数 func1 の呼び出し
           :          前に定義されているので，宣言をしなくても使用可能
    }
```

図 6-2 関数の使用宣言

C言語のプログラムは関数の集まりとして構成されますから，このような関数の呼び出し関係は，そのプログラムを解析する上で非常に重要です．関数はできるだけ明確に宣言をしておいた方が，あとでデバッグするときには役に立ちます．

## ■値の渡し方

関数へ値を渡すには**引数**を用います．この引数の渡し方には，次に示す2種類があります．

① 変数の値そのものを渡す(call by value)
② 変数のアドレスを渡す　(call by reference)

他のプログラミング言語では，②の変数のアドレスを渡す方法が一般的です．つまり，呼ばれた関数で変数そのものを書き換えてしまうことになります．これに対して①では，変数の値がコピーされて渡されます．つまり，呼ばれた関数でその値をいくら書き換えても呼び出した側の変数の値は変わりません．

仮引数の数とデータ型は，関数呼び出しでの実引数と一致している必要があります．また，仮引数で宣言できる記憶クラスは「auto」または「register」のみです．

以下では，この2つの引数の渡し方について解説します．

### ― 変数の値を渡す(call by value) ―

変数の値を渡す場合は，次ページの**図 6-3**のように行います．

関数 func が呼ばれると，関数 main での変数 x，y，z の値は，関数 func の変数 a，b，c としてコピーされスタック上に積まれます．func の実行時には，変数 a，b，c がスタックから取り出され演算が実行されます．この方法では，x，y，z の値が a，b，c という変数にコピーされて渡されるので，たとえば a の内容が関数 func のなかで変わっても x はそのままスタック上に残っています．

```
main()
    {
    int    x, y, z, p;

    p = func(x, y, z);
    printf("p ---> %d¥n", p);

    }

int    func(a, b, c)
    int    a, b, c;
    {
    int    d;

    d = a + b + c;

    return(d);
    }
```

実引数 x, y, z が関数 func の仮引数 a, b, c にコピーされる

仮引数の宣言(実引数と同じ名前である必要はない)

仮引数を使った演算

返値として演算結果を戻す

z　y　x

c　b　a

関数 func の呼び出し

関数 func の実行

c　b　a

それぞれの変数のコピーがスタック上に積まれる

コピーされた変数が取り出されて使われる

c　b　a　z　y　x

関数 func からリターンしても変数 x, y, z はスタック上に残っている

スタック領域

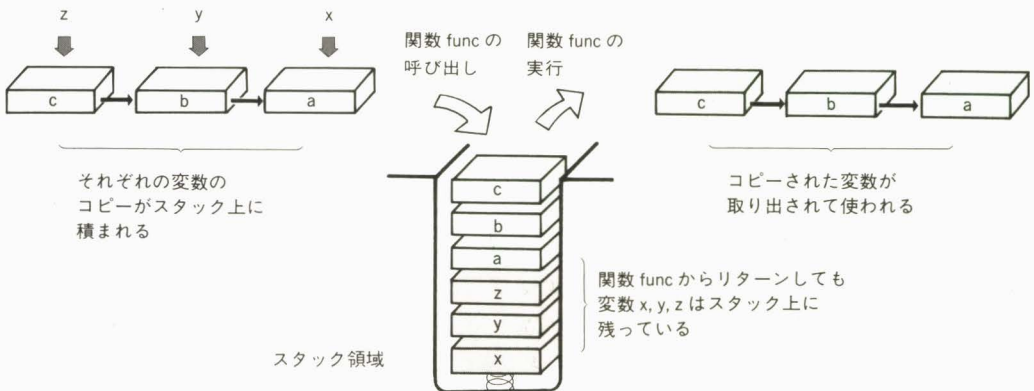



図 6-3　引数を値で渡す場合

## — 変数のアドレスを渡す(call by reference) —

変数のアドレスを渡す場合は，**図 6-4** のようになります．

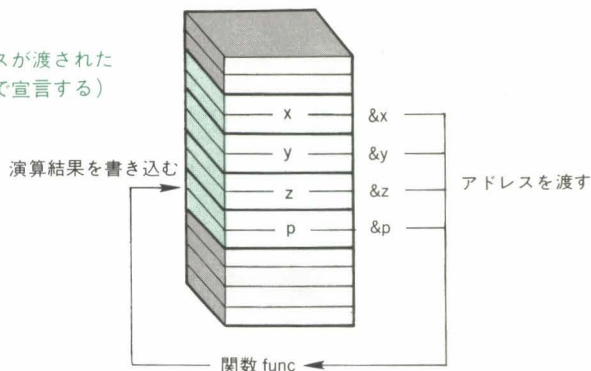
```
main()
    {
    int     x, y, z, p;
            :
    func(&x, &y, &z, &p);
    printf("p ---> %d¥n", p);
            :
    }

void    func(a, b, c, d)
    int     *a, *b, *c, *d;
    {
            :
    *d = *a + *b + *c;
            :
    }
```



図 6-4 引数をアドレスで渡す場合

図 6-3 とまったく同じことをやらせるのに今度はすべてポインタ変数を使ってみました. 関数 main から関数 func に渡される値は, 変数 x, y, z のポインタ(アドレス)です. 実値ではなく変数のあるアドレスが渡されるので, 関数 func での引数はポインタ変数として宣言されなくてはなりません. またこの場合, 関数 func で「*a」の値を変更すると, func から main に戻ったとき変数 x の値は変わってしまいますので注意してください.

もっともこのことを使って, 呼び出した関数 func から値をもらうことができます. たとえば, 関数 main の変数 p のように「func から返してもらう値を入れる変数」を作っておき, そのポインタを func に渡すことで, func から返ったときには, その変数に必要なレスポンスが入っているという使い方ができます. この方法は, return 文と違い複数の値を返すこともできるので, よく使われる方法です.

以上が関数どうしでの値の受渡しですが, 引数を使わずに**グローバル変数**で値を受け渡すこともできます(「4.5 変数の有効範囲」を参照). ただし, この方法を使ったプログラムでは, 関数自身の独立性が保てないため汎用性がなくなります. このような受渡しをすると, ある関数だけを独立して別のプログラムに組み込むときに苦労をすることになるので, できるだけ引数を用いて行う方が賢明です.

# 6.2 再帰

C言語の関数は，再帰呼び出し(recursive call)を行うことができます．つまり，関数のなかで自分自身を呼び出せるわけです．よくゲームなどで使われる手法ですが，通常はあまり多く使う場面はありません．しかし，アルゴリズムによっては，この手法を使った方がより簡潔にその操作を記述できる場合があります．

## ■再帰の考え方

再帰は関数の機能というよりも，プログラミングの手法の1つとして考えることができます．

関数が再帰呼び出しを行うと，そのなかで使われている変数がスタック領域に積まれていきます．そして，retun文が実行されるたびにスタックから変数が復帰します．このように，何度も再帰呼び出しが行われると多くのスタック領域が必要になるので，あまり効率的な手法とはいえません．しかし，再帰呼び出しを利用すると，プログラムを簡潔に，そしてわかりやすく記述することが可能です．とくに，第8章で取り上げる木構造のデータを検索する場合などに使うと便利です．

非常に多くの再帰呼び出しを行う場合は，無限ループなどに陥ってしまいスタック領域をオーバーフローさせないように注意してください．

## ■サンプルプログラム

ここでは，再帰のサンプルプログラムとして，「階乗計算」をやってみましょう．階乗の計算は，もちろん再帰を用いなくても可能ですが，再帰を使った方がスマートに書けます．

以下の**リスト6-1**にプログラムとその実行結果を示します．

```
 1: main()
 2:     {
 3:     char    s[10];
 4:     int     i;
 5:
 6:     unsigned long    fact(); ………関数factの宣言
 7:
 8:     while(1)
 9:         {
10:         printf("Number ?: ");
11:         gets(s);
12:
13:         if(s[0] == '¥0')     break;
14:         i = atoi(s); ………文字列をint型の整数に変換
```

```
15:
16:             printf("      %2d! -----> %ld¥n", i, fact(i));
17:             }
18:         }
19:
20: unsigned long     fact(n) ·········unsigned long 型の返値を返す関数 fact の定義
21:         int     n;
22:         {
23:         if(n == 0)      return(1);
24:         else            return(n * fact(n-1));
25:         }                           └──→ 再帰呼び出しを行う
```

[実行結果]

```
A>test
Number ?: 0
        0! -----> 1
Number ?: 1
        1! -----> 1
Number ?: 5
        5! -----> 120
Number ?: 8
        8! -----> 40320
Number ?: 13 ·········14 以上の階乗は, unsigned long 型で表わせる範囲を越えてしまうので正しい結果は出ない
       13! -----> 1932053504
Number ?:

A>
```

リスト 6-1　階乗を求めるプログラム

ここで, fact(3)を計算するときのスタックの動作を図 6-5 で見てみましょう. この図でわかるように, C 言語での再帰呼び出しは, スタックをうまく利用することによって実現されています.

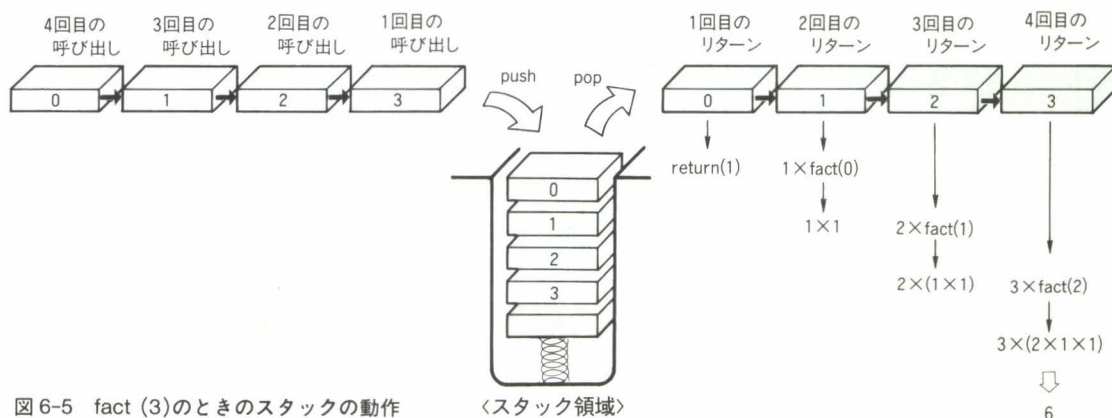



図 6-5　fact (3)のときのスタックの動作

# 6.3 入出力以外でよく使う標準関数

この節では，最初からコンパイラに付属してくる標準関数のなかでよく使うメモリ管理や文字列操作などの関数をいくつか紹介します．ここではよく使われる関数について解説しますので，それ以外の関数については APPENDIX を参照してください．

なお，入出力関数については，次章で取り上げます．

## ■メモリ管理関数

C言語でプログラムを組む場合，最も頻繁に使われる関数の1つにメモリ管理のための関数があります．これらの関数は，たとえば，アプリケーション・プログラムを書く場合，プログラム中で一時的にワークエリアを確保するときに使われます．

配列による宣言では，コンパイル時に固定したデータが割り当てられてしまいますが，メモリ管理関数を使うと実行時に必要なメモリが動的に確保できるので，効率的なメモリの活用が可能になります．また，メモリ管理関数のなかには，確保したメモリ領域が使用済みになった場合，他の用途で使えるようにそのエリアを解放する関数も用意されています．

**表6-3**にメモリエリアを確保する malloc 関数とそのエリアを解放する free 関数の仕様を示します．

| 関数名 | 書式 | 返値 | 機能 |
|---|---|---|---|
| malloc | char *malloc (size);<br>unsigned size; | NULL＝割り当てに失敗<br>NULL≠割り当てたメモリエリアの先頭アドレスのポインタ | SIZE バイトのメモリ領域を割り当てる |
| free | void free (ptr);<br>char *ptr; | なし | malloc で割り当てたメモリを解放する |

表6-3　メモリ管理関数

以下では，メモリ管理関数の使い方を手順を追って解説していきます．

### (1) エリアの確保

メモリ上に 32K バイトのエリアを確保したい場合は，以下のように行います(**図6-6**)．

```
char      *mp;  ………確保されたエリアの先頭アドレスを受け取る変数
    ⋮
if(NULL == (mp = malloc(32 * 1024)))
     {                                  確保したいエリアをバイト単位で渡す
     printf("¥7***** No memories *****¥n");
     exit(0);
     }
     ⋮
```

図 6-6　エリアの確保

malloc 関数の引数には，確保したい領域の大きさをバイト数で渡します．また返値は，char へのポインタとして返ってきますので，もし確保したい領域を short や long として利用したい場合はキャスト変換を行います(「2.4 キャスト演算子」を参照)．たとえば short として扱いたい場合は，malloc の行を以下のように変更します．

```
if (NULL == (mp = (short *)malloc (32 * 1024)))  … 32K バイトのメモリ領域を short 型の
                                                    変数の集まりとして使えるように確保
```

もちろん，この場合は変数 mp を short へのポインタとして宣言しておかなければなりません．次ページの図 6-7 に，確保した領域の扱いを示します．

また，malloc が返してくる値が NULL である場合は，その領域が確保できなかったことを示すので，図 6-6 では exit 関数によってプログラムから抜けてしまいます．

(2)確保したエリアの利用

malloc 関数は確保したエリアの先頭のアドレスをポインタとして返しますから，確保された領域は配列としても，またポインタ変数そのものとしても使うことができます．たとえば，配列として利用する場合は，

```
c = mp[135];
```

というように参照することになります．

ここで気をつけなければならないのは，確保した領域をどのデータ型の変数で利用しているかということです．malloc で確保されたエリアはバイト単位ですから，たとえば short 型の配列で扱える要素数は malloc の引数として確保した領域のバイト数の半分になります．また，もし long 型の配列ならば 1/4 になります．

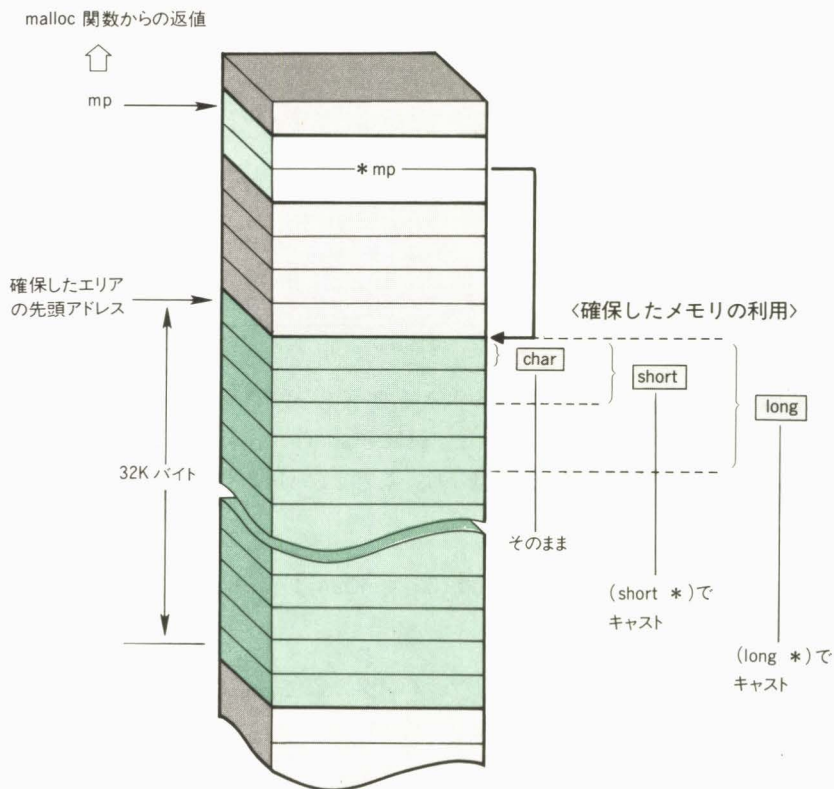



図 6-7　確保した領域の扱い

### (3)確保したエリアの解放

確保したメモリエリアを使い終わったら，別の用途でメモリが活用できるようにエリアの解放を行います．これは，以下のように free 関数を使います．

free (mp)；　…… mp を先頭アドレスとするメモリエリアを解放

確保したエリアがプログラムの最後まで使われている場合は，この操作はしなくてもかまいません．プログラムの終了と同時に自動的にこのエリアは解放されるようになっているからです．

free 関数は返値を持たず，エラーの管理をしていません．つまり，malloc で確保したエリアの先頭アドレスを引数として渡さなかった場合は，メモリ管理がおかしくなってしまいます．

## ■キャラクタコード分類／変換関数

キャラクタコード分類関数は，引数として与えられた1文字の種類を調べるための関数です．キャラクタコード変換関数は，大文字と小文字の変換などを行います．

この関数群を使うためには，「ctype .h」というヘッダファイルを取り込む必要があります．以下の**表 6-4** にキャラクタコード分類／変換関数の一覧を示します．

| 関数名 | 書式 | 返値 | 機能 |
|---|---|---|---|
| isalnum | int isalnum (c);<br>int c; | 0≠アルファベットまたは数字である<br>0＝アルファベットまたは数字でない | アルファベットまたは数字か，そうでないかを調べる |
| isalpha | int isalpha (c);<br>int c; | 0≠アルファベットである<br>0＝アルファベットでない | アルファベットかどうかを調べる |
| isascii | int isascii (c);<br>int c; | 0≠ASCII 文字である<br>0＝ASCII 文字でない | ASCII 文字(0x00～0x7F)かどうかを調べる |
| iscntrl | int iscntrl (c);<br>int c; | 0≠コントロール文字である<br>0＝コントロール文字でない | 制御文字(0x20より小さいコード)かどうかを調べる |
| isdigit | int isdigit (c);<br>int c; | 0≠10 進数値である<br>0＝10 進数値でない | 0～9 までの 10 進数値がどうかを調べる |
| islower | int islower (c);<br>int c; | 0≠小文字である<br>0＝小文字でない | 小文字かどうかを調べる |
| isprint | int isprint (c);<br>int c; | 0≠表示可能文字である<br>0＝表示可能文字でない | 画面(プリンタ)に出力可能な文字かどうかを調べる |
| ispunct | int ispunct (c);<br>int c; | 0≠句読点文字である<br>0＝句読点文字でない | 句読点(ピリオドなど)かどうかを調べる |
| isspace | int isspace (c);<br>int c; | 0≠空白文字である<br>0＝空白文字でない | 空白文字(タブやスペースなど)かどうかを調べる |
| isupper | int isupper (c);<br>int c; | 0≠大文字である<br>0＝大文字でない | 大文字かどうかを調べる |
| isxdigit | int isxdigit (c);<br>int c; | 0≠16 進数値である<br>0＝16 進数値でない | 0～9, A～F, a～f の 16 進数値であるかどうかを調べる |
| toascii | int toascii (c);<br>int c; | ASCII 文字 c を返す | ASCII 文字へ変換する |
| tolower | int tolower (c);<br>int c; | 小文字 c を返す | 大文字であれば小文字に変換する |
| toupper | int toupper (c);<br>int c; | 大文字 c を返す | 小文字であれば大文字に変換する |

**表 6-4 キャラクタコード分類／変換関数**

この表でわかるように，キャラクタコード分類関数はすべて「is～」で始まっており，返値もその文字が該当しない場合は0，該当する場合には0以外が返るというように，仕様が統一されています．したがって，これらの関数は，if文などで**図6-8**のような使い方ができます．

```
    ⋮
if(isalpha(cx))  ………if文の判定は，「0のとき偽」，「0以外のとき真」となる
    printf("cx is alphabet-string.¥n");  ………変数cxがアルファベットである場合
else
    printf("cx is not alphabet.¥n");  ………変数cxがアルファベットでない場合
    ⋮
```

図6-8　isalpha関数の使用例

また文字の変換関数は，「to～」で始まり，同じようにその仕様が統一されています．これらの関数は，文字の判定を分類コードのテーブルを調べることによって行いますので，非常に高速です（「ctype.h」ヘッダファイルの中身を参照してください）．

なお，キャラクタコード分類／変換関数は，すべて「ctype.h」というヘッダファイルのなかで**マクロ定義**されている関数です．C言語の処理系によっては，「tolower」，「toupper」の変換関数に，ライブラリ関数版が用意されているものもあります（関数とマクロの使い分けについては，「9.1 プリプロセッサ」を参照のこと）．

## ■ 文字列操作関数

文字列操作関数は，ヌル文字で終わるC言語の文字列に対して各種の操作を行います．これらの関数を使うには，「string.h」というヘッダファイルを取り込みます．**表6-5**に文字列操作関数の一覧表を示します．

これらの関数のなかで，strlenとstrcmp関数の使用例を以下の**リスト6-2**に示します．

なお，ヌル文字で終了しない文字列を操作する関数として，バッファ操作関数（memcmp，memcpyなど）が用意されている処理系もあります．くわしくは，APPENDIXを参照してください．

| 関数名 | 書式 | 返値 | 機能 |
|---|---|---|---|
| strcat | char ＊strcat (s1, s2) ;<br>char ＊s1 ;<br>char ＊s2 ; | s1 のポインタを返す | s1 に s2 をつなげ，ヌル文字を付加する |
| strcmp | int strcmp (s1, s2) ;<br>char ＊s1 ;<br>char ＊s2 ; | 0 より小…s1＜s2<br>0 …s1＝s2<br>0 より大…s1＞s2 | s1 と s2 を辞書順で比較する |
| strcpy | char ＊strcpy (s1, s2) ;<br>char ＊s1;<br>char ＊s2 ; | s1 のポインタを返す | s2 をヌル文字を含めて，s1 が示すポインタの位置にコピーする |
| strlen | int strlen (s) ;<br>char ＊s ; | s の長さをバイト数で返す | s のヌル文字を含まない文字数をバイト単位で返す |
| strncat | char ＊strncat (s1, s2, n) ;<br>char ＊s1;<br>cha ＊s2 ;<br>unsigned int n ; | s1 のポインタを返す | s1 に s2 の n 文字目までをつなげヌル文字を付加する．n が s2 の長さより大きい場合は s2 の長さとする |
| strncmp | int strncmp (s1, s2, n) ;<br>char ＊s1 ;<br>char ＊s2 ;<br>unsigned int n ; | 0より小…s1＜(s 2 の n 文字目まで)<br>0 …s1＝(s 2 の n 文字目まで)<br>0より大…s1＞(s 2 の n 文字目まで) | s1 とs2 の n 文字目までを辞書順で比較する |
| strncpy | char ＊strncpy (s1, s2, n) ;<br>char ＊s1 ;<br>char ＊s2 ;<br>unsigned int n ; | s1 のポインタを返す | s2 の n 文字目までを s1 が示すポインタの位置にコピーする |

表 6-5 文字列操作関数

```
 1: #include    <string.h>
 2:
 3: static char    puzzle[] = "reference";   ………キーワードは配列 puzzle にしまっておく
 4:
 5: main()
 6:     {
 7:     int     i, j, k;
 8:     char    str[20];                     ← キーワードの文字数だけ繰り返す
 9:
10:     for(i = 0 ; i < strlen(puzzle) ; i++)
11:         {
12:         printf("¥n(%d) Input String : ", i+1);
13:
14:         for(j = 0 ; j < i ; j++)          } ヒントの表示
15:             printf("%c", puzzle[j]);      }
```

```
16:
17:         k = strlen(puzzle) - j;
18:         while(k--)    printf("_");
19:
20:         printf("¥n                 ? : ");
21:
22:         gets(str);………文字列を標準入力から取り込む
23:         if(!strcmp(puzzle, str))………2 つの文字列を比較する
24:             {
25:             printf("    That's right!¥n");  ┐
26:             break;                           ├ 正解の場合
27:             }                                ┘
28:
29:         printf("    No, retry!¥n");………誤りの場合
30:         }
31:     }
```

[実行結果]

```
A>test↵

(1) Input String : _________
                 ? : expensive↵ ………9 文字の文字列をとりあえず入れてみる
    No, retry!

(2) Input String : r________
                 ? : recognize↵ ………「r」で始まる文字列であることがわかった
    No, retry!

(3) Input String : re_______
                 ? : recommend↵ ………「re」までは合っていたのだが……
    No, retry!

(4) Input String : ref______
                 ? : reflector↵
    No, retry!

(5) Input String : refe_____
                 ? : reference↵ ………やっと正解
    That's right!

A>
```

リスト 6-2　strlen 関数と strcmp 関数の使用例

## ■ データ変換関数

データ変換関数は，文字列を数値に変換します．これらの関数は，gets 関数などを使って文字列として入力したものを数値として利用したい場合などによく使われます．以下の**表 6-6** にデータ変換関数の一覧表を示します．

| 関数名 | 書式 | 返値 | 機能 |
|---|---|---|---|
| atof | double atof (string);<br>char ＊string; | 0≠double に変換した値<br>0＝変換不可 | 文字列を double 型の浮動小数点数に変換 |
| atoi | int atoi (string);<br>char ＊string; | 0≠int に変換した値<br>0＝変換不可 | 文字列を int 型の整数に変換 |
| atol | long atol (string);<br>char ＊string; | 0≠long に変換した値<br>0＝変換不可 | 文字列を long 型の整数に変換 |

**表 6-6 データ変換関数**

これらの関数を使う場合，atof 関数は「math.h」，atoi／atol 関数は「stdlib.h」ヘッダファイルを取り込んでから使用します(ヘッダファイルは異なる処理系がありますので，APPENDIX を参照してください).

## ■ 広域ジャンプ関数

広域ジャンプ関数には，setjmp と longjmp 関数があります．goto 文によるジャンプが関数内に限られるローカルジャンプであるのに対して，これらの関数を使うとまったく違う関数の間でグローバルにジャンプを行うことができます．

まず，この2つの関数の仕様を示します(**表 6-7**)．関数を使う上で取り込む必要があるヘッダファイルは「setjmp.h」です．

| 関数名 | 書式 | 返値 | 機能 |
|---|---|---|---|
| longjmp | void longjmp (env, value);<br>jmp_buf env;<br>int value; | なし | setjmp 関数により env にセーブした環境を復活し，対応する setjmp に制御を移す．このとき，value 値は setjmp の返値として渡される |
| setjmp | int setjmp (env);<br>jmp_buf env; | 0＝スタック環境をセーブ<br>0≠longjmp の value 値 | スタック環境を env にセーブする |

**表 6-7 広域ジャンプ関数**

ここで，「jmp_buf」は，setjmp.h に定義されているレジスタをストアするするための配列です．

これらの関数の使い方は多少難解ですが，たとえばプログラム中のある関数でエラーが起こった場合に，プログラムの最初に戻って処理を終了したいというときなどに役に立ちます．以下にサンプルプログラムを示します(**リスト 6-3**)．

```
 1: #include     <setjmp.h>
 2: jmp_buf     mx;
 3:
 4: main()
 5:     {
 6:     if(0 != setjmp(mx))
 7:         {
 8:         printf("Interrupted. Exit prcess.¥n");
 9:         exit(1);
10:         }
11:     proc0();
12:     }
13:
14: proc0()
15:     {
16:     int     i;
17:     for(i = 0 ; i < 200 ; ++i)
18:         {
19:         proc1(i);
20:         printf("Number = %d¥n");
21:         }
22:     }
23:
24: proc1(j)
25: int     j;
26:     {
27:     if(j == 100) longjmp(mx,-1);
28:     }
```

レジスタ，スタックなどを保存しておくための配列の宣言
jmp_buf は，setjmp.h に定義されているデータ型

すべての環境を mx にセーブ(返値は 0 を返す)

return 文を使うと proc 0 関数
にしか戻ることはできない

setjmp 関数の返値となる

mx にセーブしたすべての環境を呼び戻し，
setjmp に戻る

［実行結果］

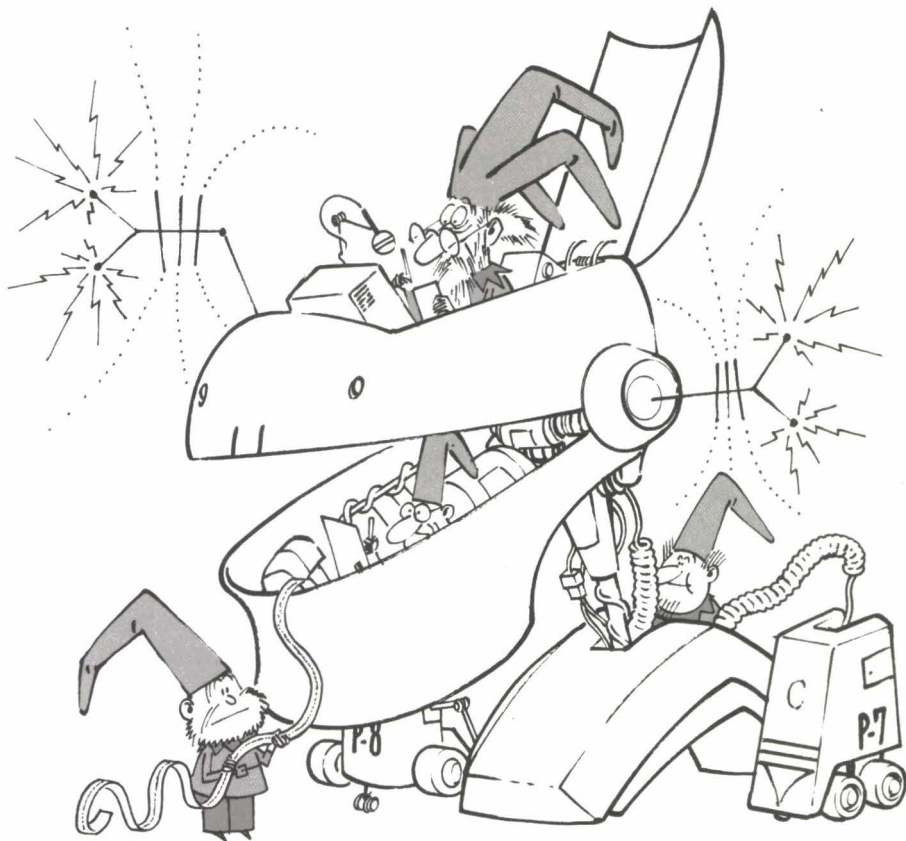
```
A>test
Number = 97
Number = 98
Number = 99
Interrupted. Exit prcess.

A>
```

リスト 6-3　setjmp 関数と longjmp 関数の使用例

このプログラムでわかるように，return 文では呼び出した関数にしか戻ることができませんが，setjmp／longjmp 関数を使うと，呼び出し関係を無視して制御を移すことができます．使い方によっては非常に便利ですが，あまり多用するとプログラムの構造が非常にわかりにくくなります．

## ■ その他の関数

このほかよく使われる関数として，sin，cos などの数値演算のための関数群やソート関数があります．数値演算用の関数は，その使い方が簡単なので APPENDIX にまとめて示しました．また，ソート関数については，「5.3 関数とポインタ」のところで qsort 関数の使い方を取り上げています．

# 第7章
# 入出力とファイル操作

BASIC や FORTRAN などの言語は，入出力のための「文」や「コマンド」を持っており，それを使うことによってファイルのオープン，クローズ，読み書きを行います．つまり，言語仕様の一部として，ファイル入出力機能が備わっています．しかし，C 言語ではこのようなプログラムの手足ともいえる構文がいっさいありません．C 言語は，これらの機能をすべてライブラリ関数として用意しています．このように非常にコンパクトな言語仕様のおかげで，C 言語はパソコンなどの小さなシステムにも早くから移植されてきたのです．

また，C 言語は UNIX というオペレーティング・システムの影響を強く受けており，その流儀にしたがった「ファイルの概念」の理解が必要です．この概念は，ファイルというよりもコンピュータの入出力のすべてに対して，まったく同じようにアクセスを可能にするという考え方から生まれてきたものです．これは MS-DOS などでも行われているファイル管理の基礎となるものですから，ぜひ覚えておいてください．

# 7.1　高水準入出力関数

ファイル入出力関数には，**高水準入出力関数**と**低水準入出力関数**の2種類があります．この2つの入出力関数には，以下の**表7-1**のような違いがあります．

| | 書式付き入出力 | バッファを介した入出力 | OSとの関係 |
|---|---|---|---|
| 高水準入出力関数 | 可能 | 可能 | OSに依存しない |
| 低水準入出力関数 | 不可能 | 不可能 | OSに依存 |

**表7-1　高水準入出力関数と低水準入出力関数の違い**

高水準入出力関数群は，はじめはより便利な入出力のためのものでしたが，最近では「どんなOS上でもまったく同じように扱える標準インターフェイス関数」としての役割の方が多くのウエイトを占めてきています．したがって，どんなアプリケーションを書くにしろ，高水準入出力関数を使うのが理想といえます．逆に低水準入出力関数は，オペレーティング・システムに依存したきめ細かな制御を行いたい場合に使われます．

なお，この節では高水準入出力関数について取り上げ，次節で低水準入出力関数を紹介します．

高水準入出力関数は，以下のように分類できます．

① ファイルのオープン／クローズ関数
② バイト入出力関数
③ 文字列入出力関数
④ フォーマット化入出力関数
⑤ ファイル操作関数
⑥ ランダムアクセス関数

以降では，入出力関数の一覧表†を示していきます．ここで紹介する関数を使用するためには，あらかじめ「stdio.h」ファイルを取り込んでおかなければなりません．このファイルには，関数の使用に際して必要な定数や構造体の定義などが含まれています(APPENDIXで各処理系のstdio.hファイルの比較をしているので参照のこと)．

また，これらの関数を使ったプログラムは，「7.3 ファイル操作の実際」で取り上げます．

---

† ここでは，多くの処理系で標準的に用意されている入出力関数を紹介する．ただし，処理系によっては，引数や返値の扱いが異なるものもあるので実際の使用にあたってはマニュアルを参照すること

## ■ファイルのオープン／クローズ関数

ファイルは読み書きを行う前にオープンし，使い終わったらクローズするというのがその扱いの基本です．以下の**表7-2**にファイルのオープン／クローズのための関数を示します．

**表7-2**のなかに出てくる「FILE」は，stdio.hファイルに定義されているファイル管理のための構造体です．このFILE構造体については，以降でくわしく取り上げます．

ファイルをオープンする場合は，そのファイルの扱いを「mode」によって指定します．「mode」の指定は，C言語の処理系やOSにより異なりますから注意してください．以下では，Microsoft C Compilerの場合を紹介しておきます(**表7-3**)．

| 関数名 | 書式 | 返値 | 機能 |
|---|---|---|---|
| fclose | int fclose (fp) ;<br>FILE ＊fp ; | 0　＝クローズに成功<br>EOF＝エラー | fpで示すファイルをクローズする |
| fopen | FILE ＊fopen (file, mode) ;<br>char ＊file ;<br>char ＊mode ; | NULL≠ファイル構造体へのポインタ<br>NULL＝エラー | fileで示すファイルをmodeの状態でオープンする |
| freopen | FILE ＊freopen (file, mode, fp) ;<br>char ＊file ;<br>char ＊mode ;<br>FILE ＊fp ; | NULL≠ファイル構造体へのポインタ<br>NULL＝エラー | fpで示すファイルをクローズし，新たにfileで示されるファイルをmodeの状態でオープンし，fpに割り当てる |

表7-2　ファイルのオープン／クローズ関数

| mode | 動作 | ファイルが存在しない場合 | ファイルが存在した場合 |
|---|---|---|---|
| "r" | 読み込み (read) | エラー | —— |
| "w" | 書き出し (write) | 作成する | 内容は失われる |
| "a" | 追加書き出し (append) | 作成する | ファイルの最後から追加 |
| "r+" | 読み込み/書き出し | エラー | —— |
| "w+" | 読み込み/書き出し | 作成する | 内容は失われる |
| "a+" | 読み込み/追加書き出し | 作成する | ファイルの最後から追加 |

上記のモードに加えて以下の変換モードを指定する

| modeに追加する文字 | 意味 |
|---|---|
| t | テキストモードでオープンする(入力時:CR-LF→LFに変換/出力時:LF→CR-LFに変換) |
| b | バイナリモードでオープンする(上記の変換を行わない) |

〈注意〉 • UNIXでは，ファイルはすべて同じように扱われるので，この区別はない．

表7-3　ファイルをオープンする際のmode (Microsoft C Compilerの場合)

このようにオープンする際のモードは，そのファイルの扱いによって何通りにも分かれていますので必要に応じて選択します．とくに，「w」や「w＋」を指定した場合は，ファイルが存在するとその内容は失われてしまいますから注意してください．

また，テキストファイルとバイナリファイルの扱いについては，「7.3 ファイル操作の実際」で取り上げます．

## ■バイト入出力関数

1バイト単位で入出力を行う関数です．ファイルに対して読み込み／書き出しを行うものと標準入出力に対して読み込み／書き出しを行うものがあります．以下の**表7-4**に，バイト入出力関数の一覧を示します．

| 関数名 | 書式 | 返値 | 機能 |
|---|---|---|---|
| fgetc | int fgetc(fp);<br>FILE ＊fp | 読み込んだ文字 | fpが示す位置から1文字読み込み，ファイルポインタを1つ進める(関数) |
| fputc | int fputc(c, fp);<br>int c;<br>FILE ＊fp; | 書き出した文字 | fpが示す位置に変数cを書き出し，ファイルポインタを1つ進める(関数) |
| getc | int getc(fp);<br>FILE ＊fp; | 読み込んだ文字 | fpが示す位置から1文字読み込み，ファイルポインタを1つ進める(マクロ定義) |
| getchar | int getchar(); | 読み込んだ文字 | 標準入出力から1文字読み込む(マクロ定義) |
| putc | int putc(c, fp);<br>int c;<br>FILE ＊fp; | 書き出した文字 | fpが示す位置に変数cを書き出し，ファイルポインタを1つ進める(マクロ定義) |
| putchar | int putchar(c);<br>int c; | 書き出した文字 | 標準出力に変数cを書き出す(マクロ定義) |
| ungetc | int ungetc(c, fp);<br>int c;<br>FILE ＊fp; | EOF≠書き戻した文字<br>EOF＝エラー | fpが示す位置に変数cを戻す．次の読み込み処理はcから始まる |

〈注意〉
- ungetc以外の関数では，ファイルのエラーやEOF(End Of File)の判定に「ferror」関数や「feof」関数を使う．

**表7-4　バイト入出力関数**

**表7-4**でわかるように，同じ機能の関数(fgetcとgetc，fputcとputc)が，純粋なライブラリ関数とマクロ定義の2種類用意されています．これらの使い分けについては，「9.1 プリプロセッサ」でくわしく解説していますので参照してください．

## ■ 文字列入出力関数

ヌル文字で終わる C 言語の文字列の入出力を行います．以下の**表 7-5** に文字列入出力関数の一覧表を示します．

| 関数名 | 書式 | 返値 | 機能 |
|---|---|---|---|
| fgets | char *fgets (b, n, fp);<br>char *b;<br>int n;<br>FILE *fp; | NULL≠読み込んだ文字列のポインタ<br>NULL＝エラーまたは EOF | fp が示す位置から文字列を b に読み込む．文字は，<br>• n 文字目まで<br>• 改行まで<br>• ファイルの終わりまで<br>のいずれかの条件が満たされるまで読み込まれる |
| fputs | int fputs (b, fp);<br>char *b;<br>FILE *fp; | EOF≠書き出した最後の文字<br>EOF＝エラー | fp で示す位置に文字列 b を書き出す．文字列の終わりのヌル文字は書き出さない |
| gets | char *gets (b);<br>char *b; | NULL≠読み込んだ文字列のポインタ<br>NULL＝エラーまたは EOF | 標準入力から 1 行(改行まで)を b に読み込む．このとき，改行をヌル文字に置き換える |
| puts | int puts (b);<br>char *b; | EOF≠書き出した最後の文字<br>EOF＝エラー | 文字列 b を標準出力に書き出す．このとき，ヌル文字を改行に置き換える |

〈注意〉
• ファイルのエラーや EOF の判定には，ferror 関数や feof 関数を使う．

**表 7-5　文字列入出力関数**

これらの関数では，エラーや EOF (End Of File) の判定を正しく行えないことがありますから，その判定には以降で紹介する「ferror」や「feof」関数を使います．

## ■ フォーマット化入出力関数

データを指定した書式に整えて入出力を行います．この関数群を使うと，データの表示桁数の指定や右詰め／左詰めなどの指定が行えるので非常に便利です．次ページの**表 7-6** にフォーマット化入出力関数の一覧表を示します．

printf／fprintf／sprintf の 3 つの関数は， format で指定する文字列として， 次のような**表現指示文字列**を含みます．

**%[符号][0][桁数][.小数部]表現指示文字**

これらは，具体的に以下の**表 7-7** で示す文字を指定します．

| 関数名 | 書式 | 返値 | 機能 |
|---|---|---|---|
| fprintf | int fprintf (fp, format [, list]) ;<br>FILE *fp ;<br>char *format ; | 書き出した文字数 | fp で示す位置へフォーマットした文字列または数値を書き出す |
| fscnaf | int fscanf (fp, format [, list]) ;<br>FILE *fp ;<br>char *format ; | format で変換した文字列または数値の個数 | fp で示す位置からフォーマットした文字列または数値を list に読み込む |
| printf | int printf (format [, list]) ;<br>char *format ; | 書き出した文字数 | 標準出力へフォーマットした文字列または数値を書き出す |
| scanf | int scanf (format [, list]) ;<br>char *format ; | format で変換した文字列または数値の個数 | 標準入力からフォーマットした文字列または数値を list に読み込む |
| sprintf | int sprintf (b, format [, list]) ;<br>char *b ;<br>char *format ; | 書き出した文字数 | 変数 b へフォーマットした文字列または数値を書き出す |
| sscanf | int sscanf (b, format [, list]) ;<br>char *b ;<br>char *format ; | format で変換した文字列または数値の個数 | 変数 b からフォーマットした文字列または数値を list に読み込む |

表 7-6 フォーマット化入出力関数

| | | |
|---|---|---|
| 符号 | + | データを右詰めで出力 |
| | − | データを左詰めで出力 |
| | 省略 | データを右詰めで出力 |
| 0 | 0 | 数値が入らない桁を 0 で埋める |
| | 省略 | 数値が入らない桁を空白で埋める |
| 桁数 | n | データを表示する桁数(符号桁を含む) |
| | 省略 | 指定されたデータを表示するのに必要な桁数 |
| 小数部 | n | 小数部を表示する桁数(数値の場合)<br>文字列中で表示する文字数(文字列の場合) |
| | 省略 | 指定されたデータを表示するのに必要な桁数 |

| 表現指示文字 | | |
|---|---|---|
| | d | 整数 (int) を 10 進数で表示 |
| | x | 整数 (int) を 16 進数で表示 |
| | o | 整数 (int) を 8 進数で表示 |
| | u | 整数 (int) を符号なしの10 進数で表示 |
| | s | 文字列として表示 |
| | c | 1 文字として表示 |
| | e | float の数値を指数形式で表示 |
| | f | float の数値を実数形式で表示 |
| | g | 上記の e または f のうち短い方を表示 |
| | l | long 型の整数 (ld, lx, lo, lu として指定) |

〈注意〉
- n は 10 進数の整数値.
- C 言語によってサポートしていない表現指示文字列があるので注意.

表 7-7 printf/fprintf/sprintf の表現指示文字列

scanf／fscanf／sscanf の3つの関数は，以下の書式で format 文字列を指定します．

**%[桁数]表現指示文字**

ここで桁数は，入力された文字列から何文字変換して取り込むかを示します．また，表現指示文字は，「g」(e または f のうち短い方を表示)がないことを除くと**表7-7**とまったく同じです．

最後にまとめとして，フォーマット化入出力関数の指定例をいくつか示しますので参照してください．

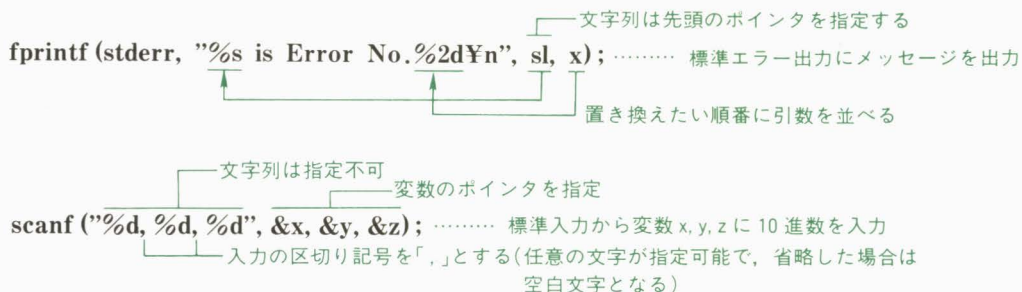



**図7-1　フォーマット化入出力関数の指定例**

## ■ ファイル操作関数

ファイル操作関数には，ファイルのエラーやエンドオブファイルの判定，入出力時に使われるバッファの操作を行う関数などがあります．以下の**表7-8**にファイル操作関数の一覧表を示します．

| 関数名 | 書式 | 返値 | 機能 |
|---|---|---|---|
| clearerr | void clearerr (fp);<br>FILE *fp; | なし | fp のエラーフラグ/エンドオブファイルフラグを0にリセットする |
| feof | int feof (fp);<br>FILE *fp; | 0＝ファイルの終わりではない<br>0≠ファイルの終わり | fp で示すファイルがエンドオブファイルかどうかを調べる |
| ferror | int ferror (fp);<br>FILE *fp; | 0＝エラーなし<br>0≠エラー | fp で示すファイルで読み込み，または書き出しエラーが起きたかどうかを調べる |
| fflush | int fflush (fp);<br>FILE *fp; | 0　＝エラーなし<br>EOF≠エラー | fp で示すバッファをファイルに書き出す |
| setbuf | void setbuf (fp, b);<br>FILE *fp;<br>char *b; | なし | デフォルトで指定されるバッファに代えて，b で指定するバッファを使用する |

**表7-8　ファイル操作関数**

これらの関数について簡単に解説しておきましょう．入出力関数を使ったファイルの読み書きでは，エラーやエンドオブファイルが発生するとそのファイルを指している FILE 構造体のフラグがそれぞれセットされます(FILE 構造体の中身については以降で解説する)．そのフラグがセットされたかどうかを調べるのが「ferror」と「feof」関数で，それぞれ stdio.h ファイルのなかでマクロ定義されています．また，フラグは一度セットされると自動的に解除されませんから，エラー処理を行った後でそのファイルを再び使いたい場合は，「clearerr」関数でエラー状態を解除します．

これまで紹介した高水準入出力関数を使ったファイルとのやりとりでは，以下の図 7-2 に示すように指定したバッファを介して行われます．

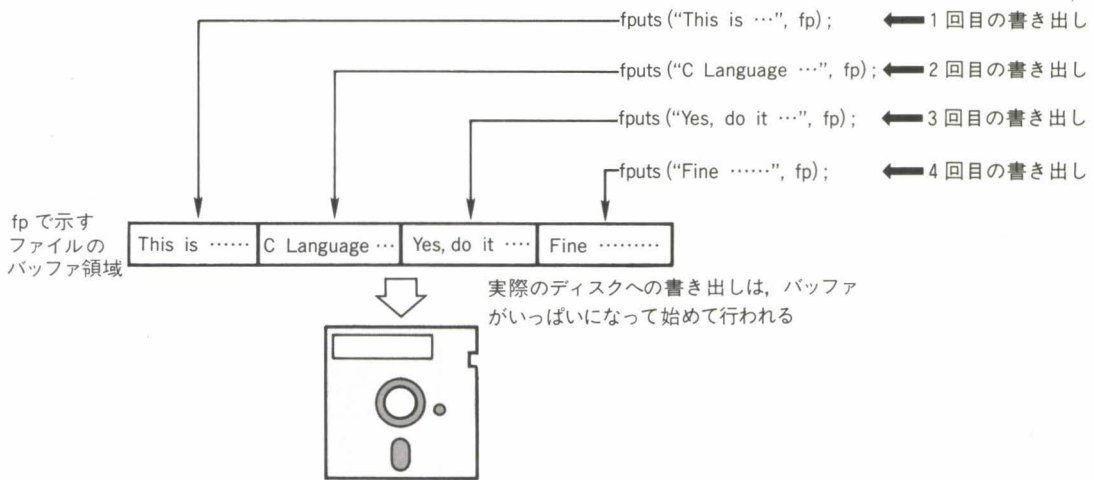



**図 7-2　バッファを介したファイルとのやりとり**

この図でわかるように，入出力関数による読み込みや書き出しは，実際にはディスクではなくバッファに対して行われます．そして，実際のディスクへの書き出しは，バッファがいっぱいになって初めて実行されます．このようにバッファリングが行われることによって，ディスクとのやりとりの効率を上げることができます．なお，こういった物理的なディスクとのやりとりはシステム側で自動的に行いますから，ユーザーはいっさい気にする必要はありません．

このようなバッファの操作を制御するのが，「fflush」と「setbuf」関数です．「fflush」は，バッファにたまったデータを強制的に書き出します．また「setbuf」は，FILE 構造体で指定されているバッファを任意の位置に変更するときに利用します．

## ■ランダムアクセス関数

ファイルのアクセスは，ファイル上の位置を表すシークポインタ(ファイルポインタ)を用いて行います．このポインタを操作するのが，以下の**表7-9**に示すランダムアクセス関数です．

| 関数名 | 書式 | 返値 | 機能 |
| --- | --- | --- | --- |
| fseek | int fseek (fp, offset, mode) ;<br>FILE *fp ;<br>long offset ;<br>int mode ; | 0＝実行終了<br>0≠エラー | fpで示すファイルポインタをmodeからoffsetバイト移動する．modeは以下のいずれかの数値を指定する<br>0……ファイルの始め<br>1……現在のファイルポインタの位置<br>2……エンドオブファイル |
| ftell | long ftell (fp) ;<br>FILE *fp ; | −1L＝エラー<br>−1L≠現在の位置 | fpで示すファイルポインタの現在位置を調べる |
| rewind | int rewind (fp) ;<br>FILE *fp ; | 0＝実行終了<br>0≠エラー | fpで示すファイルポインタをファイルの先頭に移動し，エラーフラグとエンドオブファイルフラグをクリアする |

**表7-9　ランダムアクセス関数**

これらの関数を使ったランダムアクセスの応用例は，「7.3 ファイル操作の実際」で取り上げます．

## ■入出力の手順

ファイル入出力の手順は，高水準入出力関数を使う場合でも低水準入出力関数を使う場合でもかわりません．それは，次のような手順を踏みます．

**ファイルのオープン**　……　ファイルの指定と使用宣言
↓
**ファイルの読み書き**　……　実際の使用
↓
**ファイルのクローズ**　……　ファイルの使用終了宣言

外部記憶装置のファイルを扱うには，メモリ上の値を扱うのとは異なりオープン／クローズという手続きが必要になります．現在のコンピュータの多くはすべての入出力の管理をオペレーティング・システムを通して行っており，その管理外での入出力は行ってはいけないことになっています．そこで，外部とのデータのやり取りの際には，このようなOSの手続きに沿った手順が必要となるのです．

また高水準入出力関数では，ファイルの読み書きはすべてFILE構造体を通して行うので，ファイルの物理的な構造について理解する必要はなく，その扱いは非常に楽になっています．

## ■標準入出力

C言語が生まれたUNIXという環境は，もともとマルチユーザーで使われることが前提となっています．そのような環境では，「標準入出力」を設定しておくとプログラムを書く上でたいへん便利です．これらの標準入出力デバイスは，以下の**表7-10**のように設定されています．このような各デバイスは，すべて1つのファイルとして扱うことができます．ただし，この標準入出力は，他のファイルと異なり自動的にバッファ化されません．

| 名称 | 通常割り当てられているデバイス | FILE構造体のポインタ |
|---|---|---|
| 標準入力 | キーボード | stdin |
| 標準出力 | ディスプレイ | stdout |
| 標準エラー出力 | ディスプレイ | stderr |
| 標準補助入出力 | RS-232C | stdaux |
| 標準プリンタ出力 | プリンタ | stdprn |

〈注意〉
- 標準補助入出力/標準プリンタ出力については，パソコン上のC言語でサポートされているものがある．

**表7-10　標準入出力**

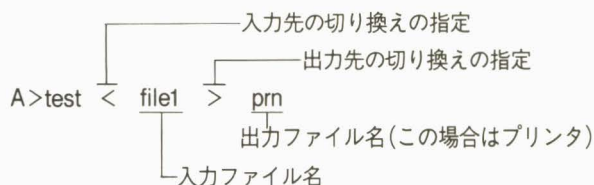
これらのファイルは，プログラムが起動すると自動的にオープンされ，プログラムの終了と同時にクローズされます．つまり，マルチユーザーの環境でも数あるデバイスのなかからとくに指定することなしに，各自の端末(コンソール)に対して入出力が行えるようになっているのです．これにより，プログラミングの際に頻繁に使われるキーボードとディスプレイの扱いが簡単になっています．

実際のプログラムでは，**表7-10**のあらかじめ設定されたFILE構造体へのポインタを使うことで，通常のファイルと同じように扱うことができます．

```
                 ┌──標準出力のFILE構造体へのポインタ
fputs("Hello", stdout) ;  …… 標準出力へ文字列を書き出す
                              (stdoutは，オープン／クローズの必要はない)
```

また，「printf」や「gets」関数などのように，はじめから標準入出力を対象とした関数も用意されています．

標準入出力を設定するもう1つのメリットは，**リダイレクト機能**と組み合わせて使うことで，入出力先の切り換えを簡単に行えることです．このリダイレクト機能は，UNIXやMS-DOSなどのオペレーティング・システムによってサポートされています(CP/Mでも，この機能をC言語の処理系自身

でサポートしているものがある）．たとえば，次のようにコマンドラインから指定することで，入出力の切り換えを行うことができます．

＊この他に「>>」を指定して，既存のファイルに追加することもできる

これは，図7-3のように実現されています．

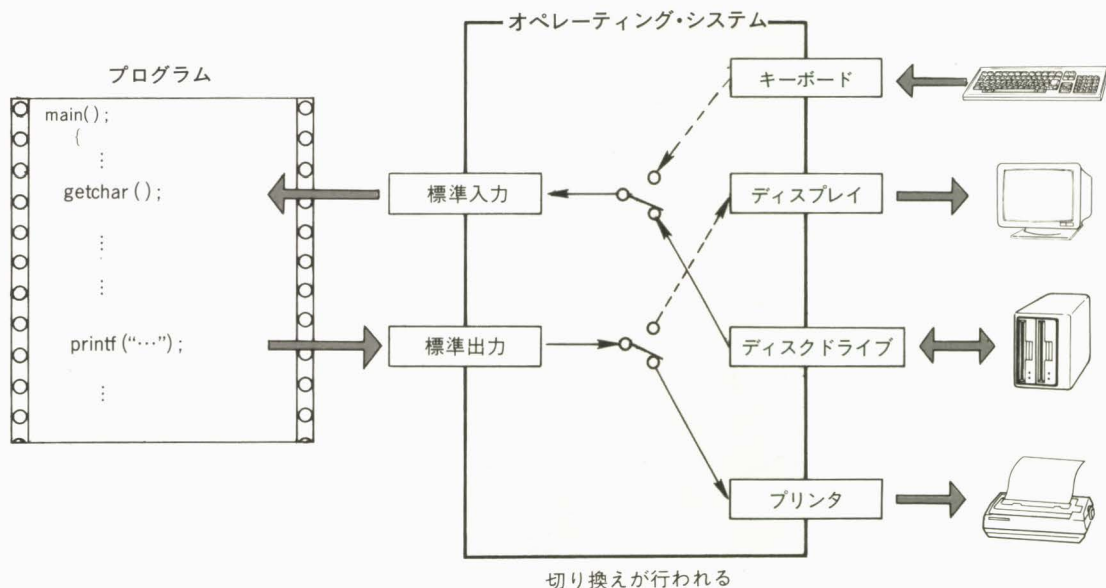



図7-3　標準入出力の切り換え

このように標準入出力を対象とするプログラムは，オペレーティング・システムのコマンドラインから簡単に入出力先を切り換えることが可能です．

なお，表7-10にある標準エラー出力は，出力先がリダイレクトされてもエラーメッセージなどは常に画面に表示されるように用意されています．

## ■ FILE 構造体

高水準入出力関数は，ファイルの入出力を管理するために，FILE 構造体という一連のデータの集まりを用います．これは，オープンされるファイルの 1 つずつにそれぞれ割り当てられます．

FILE 構造体は，「stdio . h」ヘッダファイルに定義されており，以下のような情報が詰まっています（くわしくは，第 8 章で取り上げる）．

① 現在ファイルを読み書きしている位置（ファイルポインタ）
② 現在のファイルポインタの位置からファイルの最後までのデータ数
③ データを読み書きするバッファの位置
④ ファイルにエラーが起こったかどうかを表すフラグ
⑤ 現在のファイルが何番目にオープンされたファイルかという情報

そして実際には，この構造体のポインタを扱うことによってファイルを操作することができます．先の stdin，stdout なども，実は FILE 構造体の位置を示すポインタが入っている変数なのです．

以下の図 7-4 に FILE 構造体を使ったファイル入出力の概念図を示します．

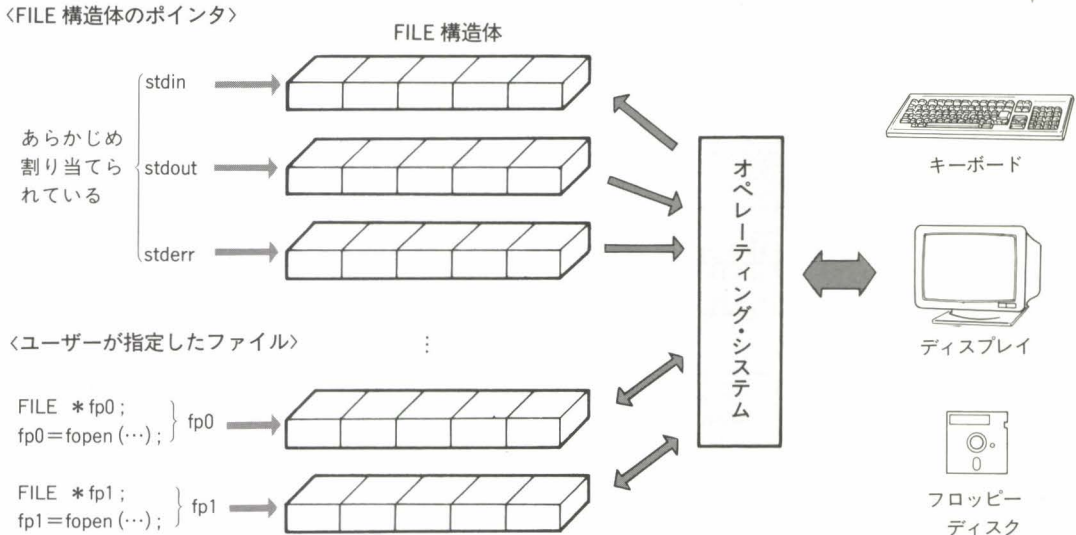



図 7-4　FILE 構造体を使ったファイル入出力の概念図

# 7.2 低水準入出力関数

低水準入出力関数は，オペレーティング・システムに依存しており，その種類も基本的なものしか用意されていません．しかし，前節で紹介した高水準入出力関数は，実は低水準入出力関数を使って実現されているのです．

この関数群は，OSに依存してしまうので，ときには，まったく互換性のない処理系同士も存在します．ですから，他の処理系やOSへの移植を意識したプログラムを書く場合はあまりおすすめできません．ただし，よりオペレーティング・システムに近い部分でのきめの細かいプログラミングが可能になりますから，より複雑なファイル処理の場合などで威力を発揮することもあります．

また低水準入出力関数はプリミティブな関数なので，自動的にバッファリングが行われたり，フォーマット入出力を行うことはできません．以下の**図7-5**に低水準入出力関数と高水準入出力関数のアクセス方法の違いを示します．

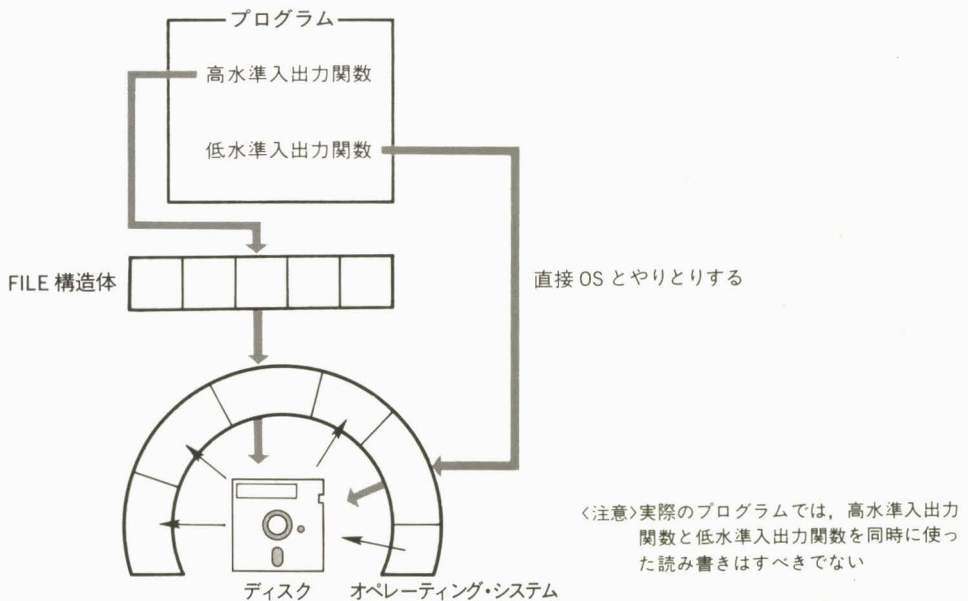



**図7-5　低水準入出力関数と高水準入出力関数のアクセス方法の違い**

## ■低水準入出力関数の一覧

まず，低水準入出力関数と高水準入出力関数を比較するために，関数の対応表を以下の**表 7-11** に示しておきましょう．

| 操作内容 | 高水準関数 | 低水準関数 |
|---|---|---|
| ファイルのオープン | fopen<br>fdopen<br>freopen | open<br>sopen<br>creat |
| ファイルのクローズ | fclose<br>fcloseall | close |
| ファイルへの出力 | fprintf<br>fputc<br>fputs<br>putc<br>puts<br>putw<br>fwrite<br>fflush | write |
| ファイルからの入力 | fscanf<br>fgetc<br>fgets<br>getc<br>gets<br>getw<br>fread | read |
| その他 | cleaerr<br>feof<br>ferror<br>fflush<br>fseek<br>ftell<br>rewind<br>setbuf<br>ungetc | dup<br>dup2<br>eof<br>lseek<br>tell |

〈注意〉
• この表は，Microsoft C Compiler を例に作成した．

**表 7-11　低水準入出力関数と高水準入出力関数の対応表**

低水準入出力関数は，ファイルを行単位や文字単位で扱うのではなく，単なるデータのブロックとして扱う場合に有用です．**表 7-11** でもわかるように，これらの関数は高水準のものよりも種類が少なく融通もききません．たとえば，ファイルへの入出力関数は，「read／write」の 2 つ関数しか用意されていません．

しかし，次節で紹介する「ディスクメモリ」のようなきめ細かな入出力の制御には低水準入出力関数の方が便利であり動作も確実です．また，バッファとのやりとりなどもプログラマ自身がすべて管理できるので，たとえば標準ライブラリにバグがある場合などでも，安心できる度合が大きくなります．

以下の**表 7-12** に，主要な 6 つの低水準入出力関数についてその仕様をまとめておきます(ここでは，Microsoft C Compiler を例にしました)．

| 関数名 | 書式 | 返値 | 機能 |
|---|---|---|---|
| open | int open (path, flag, mode) ;<br>char ＊path ;<br>int flag ;<br>int mode ; | −1≠ファイルハンドル<br>−1＝エラー | path で指定したファイルを flag の状態でオープンする．また，mode はファイルのパーミッションを設定する |
| creat | int creat (path, mode) ;<br>char ＊path ;<br>int mode ; | −1≠ファイルハンドル<br>−1＝エラー | path で指定したファイルを作成する．mode はファイルのパーミッションを設定する |
| close | int close (handle) ;<br>int handle ; | 0＝クローズ終了<br>−1＝クローズ不可 | handle で指定するファイルをクローズする |
| lseek | long lseek (handle, offset, origin) ;<br>int handle ;<br>long offset ;<br>int origin ; | −1≠シークした位置<br>−1＝エラー | handle が示すファイルポインタを origin から offset だけ移動する |
| read | int read (handle, buffer, count) ;<br>int handle ;<br>char ＊buffer ;<br>unsigned int count ; | −1≠読み込んだバイト数<br>−1＝エラー | handle と結合するバッファへ count バイト読み込む |
| write | int write (handle, buffer, count) ;<br>int handle ;<br>char ＊buffer ;<br>unsigned int count ; | −1≠書き出したバイト数<br>−1＝エラー | handle が示すファイルへ buffer から count バイト書き出す |

〈注意〉
- この表は，Microsoft C Compiler を例に作成した．他の処理系では，count で指定できる数が限られているなどの制約がある場合がある．
- 各関数は，「io.h」ヘッダファイルを取り込んでから使用する．
  ただし，以下の関数はそれに加えて次のファイルを取り込む必要がある．
  open関数……fcntl.h，sys¥types.h，sys¥stat.h
  creat関数……sys¥types.h，sys¥stat.h

**表 7-12　主要な低水準入出力関数**

上記の**表 7-12** 以外の関数として，ファイルハンドルを複製する「dup」，「dup2」関数があります．これらの関数を使用すると同一のファイルを別々のファイルハンドルで操作することが可能になります．

このほか高水準入出力関数と同じように，エンドオブファイルを判定する「eof」関数や現在のファイルポインタの位置を調べる「tell」関数が用意されている処理系があります．

## ■低水準入出力関数の扱い

高水準入出力関数では，FILE構造体のポインタを扱うことによって実際のファイル操作を行いましたが，低水準入出力関数ではオペレーティング・システムで使う**ファイルハンドル**を直接扱います．このファイルハンドルには，高水準入出力関数の標準入出力のようにあらかじめ設定されているものがあります(**表7-13**)．

| | ハンドル | | ハンドル |
|---|---|---|---|
| 標準入力 | 0 | 標準補助入出力 | 3 |
| 標準出力 | 1 | 標準プリンタ出力 | 4 |
| 標準エラー出力 | 2 | | |

〈注意〉
• 標準補助入出力/標準プリンタ出力については，パソコン上のC言語でサポートされているものがある．

**表7-13　ファイルハンドル**

ここで標準入出力は，高水準入出力関数で扱うのとまったく同じように扱うことができます(リダイレクト機能なども同じように利用できます)．

プログラマがオープンしたファイルは，**表7-13**以降のハンドルが割り当てられます．そして，FILE構造体と同様にこのハンドルを扱うことによって，ファイルの読み書きや操作を行うことができるのです．なお，open関数では，新規のファイルをオープンすることはできませんので，その場合はあらかじめcreat関数でファイルを作成しておきます．ファイルをオープンするモードについては，fopen関数とほぼ同等なものが用意されています(**表7-12**では，flagとして指定する)．ここでは，Microsoft C Compilerの例を紹介しておきます(**表7-14**)．

| モード | 動作 | モード | 動作 |
|---|---|---|---|
| O_APPEND | ファイルポインタをEOFに移動 | O_RDWR | ファイルを読み書きのためにオープン |
| O_CREAT | 書き込みのためのファイルを作成 | O_TRUNC | 既存のファイルをオープンし長さを0にする |
| O_EXCL | ファイルが存在した場合エラーを返す(O_CREATと共に利用する) | O_WRONLY | 書き込み専用としてファイルをオープン |
| | | O_BINARY | バイナリファイルとしてオープン |
| O_RDONLY | ファイルを読み出しのためにオープン | O_TEXT | テキストファイルとしてオープン |

〈注意〉
• この表は，Microsoft C Compilerの場合である．モードの指定法は各処理系によってかなり異なる．
• モードで指定する各定数は，「fcntl.h」ヘッダファイルに定義されている．
• 実際に使用する場合は，各定数のビットORをとって組み合わせる．

**表7-14　オープン関数でファイルをオープンするモード**

# 7.3 ファイル操作の実際

ここでは実際に入出力関数を使ったプログラムを組んでみることにしましょう．プログラムの例題としては，これまであまり紹介する機会のなかったバイナリモードでのオープンやランダムアクセス，低水準入出力関数を使ったものを選びましたので，その使い方をよく理解してください．

なお，このプログラムはMicrosoft C Compilerで書かれていますが，標準的な関数を使用していますので多少修正を加えれば，ほとんどの処理系で動かすことができます．各処理系間の関数やインクルードファイルの違いは，APPENDIXに対応表を一覧にしてまとめていますのでそれを参考に移植をしてください．

またページ数の関係で，プログラムの解説は最低限にとどめます．これまで解説した関数の一覧表を参考に各自で解析をしてみてください．

## ■ファイルの16進数ダンプ

MS-DOSやCP/Mなどのオペレーティング・システム上のファイルは，人間の読める文字のみを扱う**テキストファイル**と，すべてのデータを扱う**バイナリファイル**に分かれます(UNIXではこのようなファイルの分け隔てはありません)．これらはオペレーティング・システムによって，明確に区別されその扱いも異なります．たとえばテキストファイルでは文字テキストが行単位で区切られて管理されているのに対して，バイナリファイルではその区切りがありません．テキストファイルとバイナリファイルでは，とくに改行を示すCR-LFの扱いが異なってきます(**表7-3**を参照)．

しかし，ファイル自身に，このファイルがテキストなのかバイナリなのかを表す明確な印はなく，ファイルを扱うユーザーが自分で管理しなくてはなりません．つまり，バイナリファイルでもテキストモードでオープンできてしまいますし，その逆も同じようにできてしまうのです．もちろん，このようなことをするとファイルの読み書きの結果がめちゃくちゃになってしまうこともあります．

C言語での具体的なファイルの区別は，たとえば高水準入出力関数のfopen関数の場合，読み込み，書き出しを行うスイッチ(〝w〟とか〝r〟など)に〝b〟を入れて〝wb〟，〝rb〟と指定することによって，ファイルをバイナリモードで扱うことができます．低水準入出力関数の場合はとくに指定しない限りバイナリファイルとして扱われます．高水準入出力関数では，この指定がないとファイルはテキストモードでオープンされます(つまり，文書ファイルとして扱われます)．

ここでは，どんなファイルでもバイナリモードでオープンして，16進数で表示するプログラムを紹介します(**リスト7-1**)．

```
 1: /*
 2:         XDUMP.C  Dump File utility
 3: */
 4:
 5: #include     <stdio.h>
 6:
 7:
 8: usage() ………プログラムの使用法を説明する関数
 9:         {
10:         puts("¥7");
11:         puts("Usage : XDUMP <FILE-NAME>");
12:         }
13:
14:
15: main(argc, argv)
16: int         argc;
17: char     **argv;
18:         {
19:         int       i, j;
20:         FILE     *fp;
21:
22:         if(2 != argc) ………引数の個数が 1 個以外はエラーとみなす
23:                 {
24:                 usage();
25:                 exit(0);
26:                 }
27:
28:         if(NULL == (fp = fopen(argv[1], "rb"))) ………ファイルをバイナリモードでオープン
29:                 {
30:                 printf("¥7Cannot open FILE : %s¥n", argv[1]);  ┐
31:                 usage();                                        │ ファイルがオープンできない場合
32:                 exit(1);                                        │
33:                 }                                               ┘
34:
35:         for(i = 0 ; i < 0x7FFF ; ++i)
36:                 {
37:                 printf("%08lX :", ftell(fp)); ………現在のファイルポインタの位置を表示
38:
39:                 for(j = 0 ; j < 16 ; ++j) ………………1 行文(16 個)のダンプを行うループ
40:                         {
41:                         int     c;
42:
43:                         c = 0x00FF & getc(fp); ………読み込んだ 1 文字の上位ビットをマスクする
44:                         if(ferror(fp))
45:                                 {
46:                                 puts("¥7>>>> Read Error ! <<<<");
47:                                 break;
48:                                 }
49:                         if(feof(fp)) break;
50:
51:                         printf(" %02X", c); ………16 進数で表示
52:                         }
53:
```

```
54:          printf(" : %08lX¥n", ftell(fp) - 1);……最終アドレスを表示
55:
56:          if(ferror(fp) || feof(fp)) break; ………エラーまたはファイルの終わりでループを抜ける
57:          }
58:
59:      fclose(fp);
60:      puts(" ");
61:      }
```

［実行結果］

```
A>xdump xdump.c ⏎ ………自分自身のソースファイルを実行してみる
00000000 : 2F 2A 0D 0A 20 20 20 20 58 44 55 4D 50 2E 43 20 : 0000000F
00000010 : 20 44 75 6D 70 20 46 69 6C 65 20 75 74 69 6C 69 : 0000001F
00000020 : 74 79 0D 0A 2A 2F 0D 0A 0D 0A 23 69 6E 63 6C 75 : 0000002F
00000030 : 64 65 20 20 20 20 3C 73 74 64 69 6F 2E 68 3E 0D : 0000003F
00000040 : 0A 0D 0A 0D 0A 75 73 61 67 65 28 29 0D 0A 20 20 : 0000004F
00000050 : 20 20 7B 0D 0A 20 20 20 20 70 75 74 73 28 22 5C : 0000005F
00000060 : 37 22 29 3B 0D 0A 20 20 20 20 70 75 74 73 28 22 : 0000006F
00000070 : 55 73 61 67 65 20 3A 20 44 55 4D 50 20 3C 46 49 : 0000007F
00000080 : 4C 45 2D 4E 41 4D 45 3E 22 29 3B 0D 0A 20 20 20 : 0000008F
00000090 : 20 7D 0D 0A 0D 0A 0D 0A 6D 61 69 6E 28 61 72 67 : 0000009F
000000A0 : 63 2C 20 61 72 67 76 29 0D 0A 69 6E 74 20 20 20 : 000000AF
000000B0 : 20 20 20 20 61 72 67 63 3B 0D 0A 63 68 61 72 20 : 000000BF
000000C0 : 20 20 20 2A 2A 61 72 67 76 3B 0D 0A 20 20 20 20 : 000000CF
000000D0 : 7B 0D 0A 20 20 20 20 69 6E 74 20 20 20 20 20 20 : 000000DF
000000E0 : [illegible] [illegible] [illegible] [illegible] 3B 0D 0A 20 20 20 20 46 49 4C 45 20 : 000000EF
0000[illegible]B0 : 20 20 20 20 20 20 [illegible] 0D 0A 0D 0A 20 20 20 20 69 : 000000FF
[illegible]0002C0 : 65 72 72 6F 72 28 66 70 [illegible] [illegible] 63 29 0D 0A 20 20 : 0000010F
000002D0 : 20 20 20 20 20 20 20 20 20 20 [illegible] [illegible] 20 20 20 20 : 0000011F
000002E0 : 20 20 20 20 20 20 20 20 20 20 20 20 [illegible] [illegible] 20 20 : 0000012F
000002F0 : 75 74 73 28 22 5C 37 3E 3E 3E 3E 20 52 65 [illegible] [illegible] : 0000013F
00000300 : 20 45 72 72 6F 72 20 21 20 3C 3C 3C 3C 22 29 3B : [illegible]
00000310 : 0D 0A 20 20 20 20 20 20 20 20 20 20 20 20 20 20 : 0000031F
00000320 : 20 20 62 72 65 61 6B 3B 0D 0A 20 20 20 20 20 20 : 0000032F
00000330 : 20 20 20 20 20 20 20 20 20 20 7D 0D 0A 20 20 20 : 0000033F
00000340 : 20 20 20 20 20 20 20 20 20 69 66 28 66 65 6F 66 : 0000034F
00000350 : 28 66 70 29 29 20 62 72 65 61 6B 3B 0D 0A 0D 0A : 0000035F
00000360 : 20 20 20 20 20 20 20 20 20 20 20 20 70 72 69 6E : 0000036F
00000370 : 74 66 28 22 20 25 30 32 58 22 2C 20 63 29 3B 0D : 0000037F
00000380 : 0A 20 20 20 20 20 20 20 20 20 20 20 20 7D 0D 0A : 0000038F
00000390 : 0D 0A 20 20 20 20 20 20 20 20 20 70 72 69 6E 74 66 : 0000039F
000003A0 : 28 22 20 3A 20 25 30 38 6C 58 5C 6E 22 2C 20 66 : 000003AF
000003B0 : 74 65 6C 6C 28 66 70 29 20 2D 20 31 29 3B 0D 0A : 000003BF
000003C0 : 0D 0A 20 20 20 20 20 20 20 20 69 66 28 66 65 72 : 000003CF
000003D0 : 72 6F 72 28 66 70 29 20 7C 7C 20 66 65 6F 66 28 : 000003DF
000003E0 : 66 70 29 29 20 62 72 65 61 6B 3B 0D 0A 20 20 20 : 000003EF
000003F0 : 20 20 20 20 20 7D 0D 0A 0D 0A 20 20 20 20 66 63 : 000003FF
00000400 : 6C 6F 73 65 28 66 70 29 3B 0D 0A 20 20 20 20 70 : 0000040F
00000410 : 75 74 73 28 22 20 22 29 3B 0D 0A 20 20 20 20 7D : 0000041F
00000420 : 0D 0A : 00000421

A>
```

リスト7-1　ファイルの16進数ダンププログラム

## ■ 文字列表示プログラム

このプログラムは，先と同様にファイルをバイナリモードでオープンし，実行ファイル中に入っている読める文字列を画面に表示します．このプログラムは，実行ファイルの解析に利用することができます．

また，ここでは低水準入出力関数を使っています．低水準入出力関数と高水準入出力関数のプログラムではそれほど大きな隔たりはありません．とくにその扱いが異なるのは，ファイルの識別子にFILE構造体のポインタではなく，「ファイルハンドル」を用いているということです．また，ファイルのアクセスモードの指定は，「fcntl.h」ヘッダファイルに定義してある定数を使って行います(Microsoft C Compilerの場合)．

以下の**リスト 7-2** に,実行形式のファイルから文字列を表示する「strings」プログラムを紹介します．

```
 1: /*
 2:        Strings.c   readable string reader
 3: */
 4:
 5: #include      <stdio.h>
 6: #include      <fcntl.h>
 7: #include      <sys¥types.h>  低水準入出力関数を使うために必要なヘッダファイル群
 8: #include      <sys¥stat.h>   (Microsoft C Compiler の場合)
 9: #include      <io.h>
10: #include      <ctype.h> ………63 行目の iscntrl 関数が定義されている
11:
12: #define       ACCESS_ERR      (-1) ………エラーのときの返値
13: #define       ACCESS_EOF        0  ………EOF (エンドオブファイルのときの返値)
14:
15:
16: usage() ………プログラムの使用法を説明する関数
17:        {
18:        puts("¥7");
19:        puts("Usage : strings <file-name>.");
20:        }
21:
22:
23: main(argc, argv)
24: int          argc;
25: char      **argv;
26:        {
27:        int       fd; ………ファイルハンドルを入れておく変数
28:        long      i;
29:        int       flg;
30:
31:        if(2 != argc) ………引数の個数が 1 個以外はエラー
32:            {
33:            usage();
34:            exit(0);
35:            }
```

```
36:
37:     if(ACCESS_ERR == (fd = open(argv[1], O_RDONLY | O_BINARY)))
38:         {
39:         puts("¥7***** Cannot open *****");
40:         exit(1);
41:         }
42:
43:     flg = 0;
44:
45:     for(i = 0 ; i < 0x7FFFFFFF ; ++i)
46:         {
47:         char    c;
48:         int     read_flg;
49:
50:         if(ACCESS_ERR == (read_flg = read(fd, &c, 1)))
51:             {
52:             puts("¥7**** Read Error ****¥n");
53:             break;
54:             }
55:         if(read_flg == ACCESS_EOF)     break;
56:         if((0 == c) && (flg == 0))
57:             {
58:             puts(" ");
59:             flg = 1;
60:             }
61:         if((c != 0) && (flg != 0))     flg = 0;
62:
63:         if(iscntrl(c))     continue;
64:
65:         putchar(c);
66:         }
67:
68:     close(fd);
69:     }
70:
```

読み込みのためのフラグ
バイナリモードでオープンするためのフラグ
ともに「fcntl.h」ファイルで定義されている
ビットORをとることで2つのフラグが立つ
1バイトずつ変数cに読み込む
読み込みエラーの判定
表示をする文字としない文字の判別
ファイルのクローズ

［実行結果］

```
A>strings strings.exe
MZ

 +~エJヘ!6向ｿｨｹｰ+ﾏ3ﾀ  皆鱒ｹ
+阿屈  3  ~P霧error 2000: Stack overflowe
2: Floa        - (C)Copyright Mi       ｱｺ-2
I@ヘ!
P    1 2001: Null pointer assignmen       *@t ク  鰓
                                        寔D 欺皆壱huL×u+灰
Usage : strings <file-name>.
***** C
***
```

リスト7-2　実行ファイルの文字列表示プログラム

## ■ランダムアクセス・プログラム

このプログラムは，fseek 関数を使ってファイルの任意の場所の値をランダムに読み込むプログラムです．題材としては「10進→16進変換」を，プログラムの最初で作成したディスクファイル上のテーブルを参照することによって行うものです．ここでは，fseek 関数の使い方を覚えてください．

以下の**リスト 7-3** にランダムアクセス・プログラムを示します．

```
 1: /*
 2:         Random Access test
 3: */
 4:
 5: #include      <stdio.h>
 6: #include      <stdlib.h> ················46 行目の「atol」関数が定義されている
 7:
 8: #define       S_FILE        0  ········fseek 関数でポインタを操作する基準を
 9:                                        ファイルの始めに設定
10: FILE     *fp;
11:
12: main()
13:         {
14:         int       i;
15:
16:         if(NULL == (fp = fopen("TESTFILE.DAT", "wb")))···現在いるディレクトリに「TESTFILE.DAT」
17:                 {                                        を作成し，バイナリモードでオープンする
18:                 puts("¥7Cannot open Data-File for write.¥n"),
19:                 exit(1);
20:                 }
21:
22:         for(i = 0 ; i < 0x7FFF ; ++i)  } 10 進→ 16 進変換テーブルにデータを書き出す
23:                 fprintf(fp, "%5X", i);  }
24:
25:         fclose(fp);
26:
27:
28:         /* ********* Dummy-Data written ********** */
29:
30:         if(NULL == (fp = fopen("TESTFILE.DAT", "rb")))········用意された変換テーブルをオープン
31:                 {
32:                 puts("¥7Cannot open Data-File for read.¥n");
33:                 exit(1);
34:                 }
35:
36:         for(;;)
37:                 {
38:                 char     lbuf[40];
39:                 long     j;
40:
41:                 fputs("input Data in DECIMAL : ", stdout);  } 10 進数データの入力
42:                 gets(lbuf);                                 }
43:
```

```
44:         if((lbuf[Ø] == 'E') || (lbuf[Ø] == 'e'))     break; ……先頭の文字が「e」または「E」ならば終了
45:
46:         j = atol(lbuf) * 5L; ………入力されたデータを long 型に変換
47:
48:         if(Ø != fseek(fp, j, S_FILE)) ………ファイルポインタの位置を移動
49:             {
50:             puts("¥7Seek Error !");    ┐
51:             exit(1);                   │ エラーの場合
52:             }                          ┘
53:
54:         if(1 > fread(lbuf, 5, 1, fp))
55:             {        └──fp が示すファイルから 5 バイトの長さのデータを 1 度読み，lbuf にストアする
56:             puts("¥7read error !");
57:             exit(1);
58:             }
59:         lbuf[5] = Ø; ………データの最後にヌル文字を追加
60:         printf("Data is %s in HEX.¥n", lbuf); ………変換結果の表示
61:         }
62:     fclose(fp);
63:     }
```

［実行結果］

```
A>random↵ ………TESTFILE.DAT ファイル(約170KByte)のデータを作成するのに多少時間がかかる
input Data in DECIMAL : 12↵
Data is     C in HEX.
input Data in DECIMAL : 3ØØ↵
Data is   12C in HEX.
input Data in DECIMAL : 492Ø↵
Data is  1338 in HEX.
input Data in DECIMAL : e↵ ………e または E でプログラム終了

A>
```

リスト 7-3　ランダムアクセス・プログラム

## ■ディスクメモリ

最後に，1つ大きなプログラムを紹介しましょう．ディスクメモリとは，ハードディスクやフロッピーディスクをあたかもメモリの一部のように使うシステムで，いくつかの関数から構成されます．ちょうど今はやりの RAM ディスクとまったく反対のデバイスと思えばよいでしょう．

C 言語の標準関数には，メモリエリアを割り当てる関数である「malloc」と，それを解放する「free」が用意されています．そこで，まずこのディスクメモリにも「割り付け」と「解放」の 2 つの関数を用意します．アドレスは符号付きの 32 ビット長で考えます．32 ビットあると，およそ 2000M バイト以上のデータが扱えますから，通常の使用では十分過ぎるデータ量です．

また，確保したエリアに対して，読み込み／書き出しを行う関数も用意しなくてはなりません．とりあえず，最低の機能を揃えるとすると，以下のような関数が必要になります．

① d_alloc …… ディスク上にメモリとしてのファイルを割り当てる
② d_free …… ディスク上のメモリに使われていたファイルを解放する
③ d_peek …… ディスク上のメモリから値を読む
④ d_poke …… ディスク上のメモリに値を書く

また便利な機能として，簡易型の仮想記憶をサポートする機能も設けておきます．この機能は，普段はメモリ上のエリアをアクセスするように作っておき，ある大きさを越えたら関数の方で勝手にメモリとディスクとのやりとりを行うようにします．こうすることで処理速度の向上が図れます．

以下の**図 7-6** にディスクメモリの概念図を，**図 7-7** に関数の呼び出し関係を示しておきます．

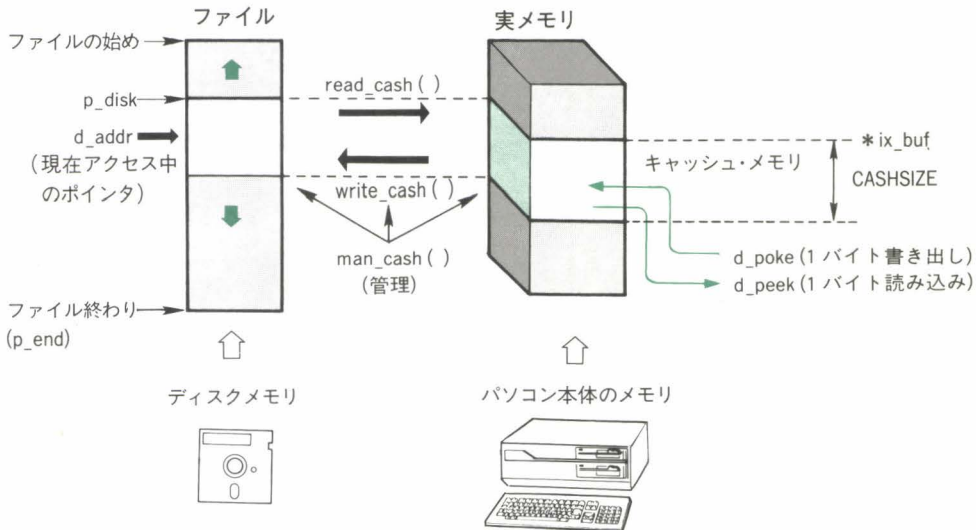



**図 7-6 ディスクメモリの概念図**

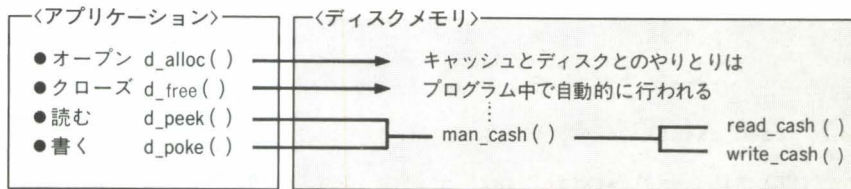



**図 7-7 ディスクメモリの関数の呼び出し関係**

多量のメモリを食うグラフィックスやデータベースのようなアプリケーションの場合，必要なメモリが線形に扱えるというのは大きなメリットです．また，エディタのようなプログラムでも，そのような要求はあるでしょう．**リスト7-4**では，最後に簡単なチェック用の書き出しプログラムを用意しましたが，各自でディスクメモリを利用した有効なアプリケーションを作ってみてください．

```
 1: /*
 2:        SOFTWAR VERTIAL-MEMORY-SYSTEMS
 3:        Created by N.Mita  at  1986/07/20
 4: */
 5:
 6: #define      UCB          0  } UNIX か MS-DOS かのスイッチ
 7: #define      MSDOS        1  }
 8:
 9: #include     <stdio.h>       }
10: #include     <sys/types.h>   } UNIX と MS-DOS に共通のインクルード・ファイル
11: #include     <sys/stat.h>    }
12: #include     <errno.h>       }
13:
14: #if UCB
15: #include     <sys/file.h>  } UNIX で必要なインクルード・ファイル
16: #endif
17:
18: #if MSDOS
19: #include     <fcntl.h>     }
20: #include     <malloc.h>    } MS-DOS で必要なインクルード・ファイル
21: #include     <io.h>        }
22: #include     <stdlib.h>    }
23: #endif
24:
25: #define      CASHSIZE     0x8000 ………メモリ内で管理するキャッシュメモリの大きさ
26: #define      SEEK_ABS     0  }
27: #define      SEEK_REL     1  } lseek 関数の origin の値
28: #define      SEEK_END     2  }
29:
30: #define      ERR_FILE     (-1)………標準関数のエラー時の返値
31: #define      OUT_SIZE     128  ………errno の拡張
32:
33: static       long    p_disk; ……現在のファイル上でのキャッシュメモリのポインタ
34: static       long    p_end;………キャッシュメモリのエンドのポインタ
35: char         *ix_buff;  ………キャッシュメモリの先頭のポインタ
36: char         fname[64];  ……読み書きするファイル名
37: int          fp_disk;  ………ファイル識別子
38:
39: int          errno; ………エラーナンバーがセットされる変数
40:
41: void    read_cash() ………ファイルからキャッシュメモリへの読み込み用の関数
42:     {
43:     if(ERR_FILE == lseek(fp_disk, p_disk, SEEK_ABS)) ………ファイル上の読み込み位置へ
44:         {                                              ポインタをシーク
45:         printf("¥7>>>>>> on READ Seek Error <<<<<<¥n");
```

```
46:          switch(errno) ………シーク時のエラーのリカバリ
47:               {
48:               case    EBADF: ………「error.h」に定義されているマクロ
49:                    printf("¥7>>>> Illigal FILE-HANDLE <<<<¥n");
50:                    break;
51:               case    EINVAL:
52:                    printf("¥7>>>> Illigal ORIGIN or pointer <<<<¥n");
53:                    printf("---- POINTER is %lx¥n", p_disk);
54:                    break;
55:               default:
56:                    printf("¥7>>>> Unknown <<<<¥n");
57:               }
58:          close(fp_disk);
59:          exit(-1);
60:          }
61:     if(ERR_FILE == read(fp_disk, ix_buff, CASHSIZE)) ….ファイルからキャッシュへの読み込み
62:          {
63:          printf("¥7>>>>>> Read error <<<<<<¥n");
64:          close(fp_disk);                            ファイルの読み込み時のエラーのリカバリ
65:          exit(-1);
66:          }
67:     }
68:
69: void     write_cash() ………キャッシュからファイルへの書き出し用の関数
70:     {
71:     if(ERR_FILE == lseek(fp_disk, p_disk, SEEK_ABS)) ……ファイルの書き出し位置にポインタ
72:          {                                              をシーク
73:          printf("¥7>>>>>> on WRITE Seek Error <<<<<<¥n");
74:          switch(errno) ………シーク時のエラーのリカバリ
75:               {
76:               case    EBADF:
77:                    printf("¥7>>>> Illigal FILE-HANDLE <<<<¥n");
78:                    break;
79:               case    EINVAL:
80:                    printf("¥7>>>> Illigal ORIGIN or pointer <<<<¥n");
81:                    printf("---- POINTER is %lx¥n", p_disk);
82:                    break;
83:               default:
84:                    printf("¥7>>>> Unknown <<<<¥n");
85:               }
86:          close(fp_disk);
87:          exit(-1);
88:          }
89:     if(ERR_FILE == write(fp_disk, ix_buff, CASHSIZE)) ………キャッシュからファイルへの
90:          {                                                 書き出し
91:          printf("¥7>>>>>> Write error <<<<<<¥n");
92:          close(fp_disk);                            書き出し時のエラーのリカバリ
93:          exit(-1);
94:          }
95:     }
96:
97: long     d_alloc(f_name, size) ………ディスクメモリへのエリアの割り付け関数
98: char     f_name[]; ………使用するファイル名
99: long     size; ………割り付ける領域の大きさ(byte)
```

```
100:      {
101:      p_end = size; ………グローバル変数にファイルサイズをとっておく
102:      strncpy(fname, f_name, 64);
103:
                                     ファイルのオープン
104: #if MSDOS
105:      if(ERR_FILE == (fp_disk = open(fname, O_CREAT | O_BINARY | O_RDWR)))
106: #endif  /* MSDOS */                   新たにファ              Read/Write
107:                                         イルを作成              モード
                                                      バイナリ(8 ビット)モード
108: #if UCB
109:      if(ERR_FILE == (fp_disk = open(fname, O_CREAT | O_RDWR)))
110: #endif  /* UCB */
111:
112:          {
113:          printf("¥7>>>>>> Cannot open DISK-MEMORY-FILE : %s <<<<<<¥n", fname);
114:          switch(errno) ………ファイルのオープン時のエラーのリカバリ
115:              {
116:              case    EACCES:
117:                  printf("¥7< Cannot ACCESS >¥n");
118:                  return(0);
119:              case    EEXIST:
120:                  printf("¥7< FILE is ALEADY exist > ¥n");
121:                  return(0);
122:              case    EMFILE:
123:                  printf("¥7< Too many open-files >¥n");
124:                  return(0);
125:              case    ENOENT:
126:                  printf("¥7< No files > ¥n");
127:                  return(0);
128:              default:
129:                  printf("¥7< Unknown ERROR >¥n");
130:                  return(0);
131:              }
132:          }
133:      if(NULL == (ix_buff = malloc((unsigned)CASHSIZE)))
134:          {
135:          printf("¥7>>>>>  memories.<<<<<<¥n");        メモリ内のキャッシュの確保
136:          return(0);
137:          }
138:      p_disk = 0; ………ファイルポインタのリセット
139:      read_cash();
140:      return(1);          これから読み書きするエリアが、キャッシュ内にあるか
141:      }                   どうかを判断し、もしなければ現在のキャッシュをメモ
142:                          リ上に書き、新しいエリアをキャッシュに読み込む
143: void    man_cash(d_addr) ………キャッシュとファイルの管理を行う関数
144: long    d_addr;
145:      {                  キャッシュ内にいるかどうかの判定
146:      if((d_addr >= p_disk) && (d_addr < (p_disk + CASHSIZE)))    return; ……
147:      else ………入っていない場合
148:          {                                                  入っていれば何も
149:          write_cash(); ………キャッシュをファイルに書く          せずに戻る
150:          p_disk = d_addr - (CASHSIZE / 2);                     新しく読ん
151:          if(p_disk < 0)                  p_disk = 0;          でくるエリ
152:          if((p_disk + CASHSIZE) > p_end)  p_disk = p_end - CASHSIZE;  アを決定す
153:          read_cash();                                          る
```

```
154:             return;          } 新しいポインタからキャッシュを読んできて返る
155:             }
156:         }
157:
158: char     d_peek(d_addr) ………ディスクメモリ上からの読み込み関数
159: long     d_addr;
160:         {
161:         if((d_addr >= p_end)||(d_addr < Ø))
162:             {
163:             errno = OUT_SIZE;               } アドレスがディスクメモリ上にあるかどうかの判断
164:             return(Ø);
165:             }
166:         man_cash(d_addr); ………キャッシュ↔ディスクの管理
167:         return(*(ix_buff + d_addr - p_disk)); ………キャッシュから値を返す
168:         }
169:
170: void     d_poke(d_addr, d_data) ………ディスクメモリ上への書き出し関数
171: long     d_addr;
172: char     d_data;
173:         {
174:         if((d_addr >= p_end) || (d_addr < Ø))
175:             {
176:             errno = OUT_SIZE;               } アドレスがディスクメモリ上にあるかどうかの判断
177:             return;
178:             }
179:         man_cash(d_addr); ………キャッシュ↔ディスクの管理
180:         *(ix_buff + d_addr - p_disk) = d_data; ………キャッシュに値を書く
181:         }
182:
183: void     d_free() ………ディスクメモリのエリアの開放
184:         {
185:         close(fp_disk);
186:         fp_disk = open(fname, O_TRUNC | S_IREAD | S_IWRITE);
187:         close(fp_disk);                └ファイルの大きさを0にする
188:         }
189:
190:
191: #define      VMSIZE       512ØØØ ………512KBのディスクメモリを作成する
192:
193: main() ………ディスクメモリの動作チェック用プログラム
194:         {
195:         char     buxx[2Ø];  (最初に512KB分のエリアを確保し，そのなかにテストデータを書き出す．次に
196:         long     i;          キーボードからの入力にしたがって任意のデータを読み込む)
197:                          ………日本語が使えない処理系では英文にする
198:         printf("現在領域確保中です¥n");  ──→サイズ
199:         if(Ø == d_alloc("TEST.TMP", VMSIZE))
200:             {                         ──→ファイル名  ──→ディスク上に領域を確保する
201:             printf("¥7**** No Area ****¥n");
202:             exit(-1);
203:             }
204:
205:         printf("¥7領域が確保されました．ダミーデータを書きに行きます¥n");
206:         for(i = Ø ; i < VMSIZE ; ++i)
207:             {
```

```
208:            d_poke(i, '0' + (i % 8)); ………ディスクメモリへのダミーデータの書き込み
209:            if(errno == OUT_SIZE)     break;
210:            }
211:
212:        printf("¥7¥7****** COMPLETED *******¥n¥n¥n");
213:        for(;;) ………入力された数値によってディスクメモリ中のデータを
214:            {           読み込んで表示する
215:            long    ax;
216:
217:            printf("INPUT ADRESS : ");
218:            gets(buxx);
219:            if('x' == buxx[0])      break;
220:            ax = atol(buxx);
221:            printf("ADRESS : %ld    VALUE : %02x¥n", ax, d_peek(ax));
222:            if(errno == OUT_SIZE)  break;
223:            printf("p_disk = %ld¥n¥n", p_disk);
224:            }
225:        d_free(); ………ディスクメモリのクローズ
226:        }
```

[実行結果]

```
A>test
現在領域確保中です
領域が確保されました．ダミーデータを書きに行きます
****** COMPLETED *******


INPUT ADRESS : 0
ADRESS : 0    VALUE : 30
p_disk = 0 ………現在のファイル上でのキャッシュメモリのポインタの位置

INPUT ADRESS : 40000
ADRESS : 40000    VALUE : 30
p_disk = 23616 ←┐
                │  ポインタの位置がキャッシュからはずれたので，
INPUT ADRESS : 20000 │  ディスクメモリから新たな読み込みが行われた
ADRESS : 20000    VALUE : 30
p_disk = 3616  ←┘

INPUT ADRESS : 20001
ADRESS : 20001    VALUE : 31
p_disk = 3616

INPUT ADRESS : 20002
ADRESS : 20002    VALUE : 32
p_disk = 3616

INPUT ADRESS : x ………x でテスト終了

A>
```

リスト 7-4　ディスクメモリ

# 第8章
# 構造体と共用体

構造体と共用体は，C言語のなかの最もC言語らしい機能の1つです．これらを用いると，複雑なデータを簡潔に記述することができ，その取り扱いは非常に楽になります．

C言語の発明者たちの書いたプログラムを見てみると，この構造体や共用体を多用して，無駄のないプログラムを書いているのがわかります．入門編では，これらのデータ型は少し難解な説明を必要とするので取り上げませんでしたが，本章でくわしく解説したいと思います．

実際に，構造体と共用体はC言語学習の要です．この機能を使いこなすためには構造体，共用体の性質を「感じ」としてつかみとって，自分のものにするということが大切です．細かなくわしい知識をいくら集めてもそれ自身はたいした意味を持ちませんし，C言語にはそんなにたくさんの「覚えるべきこと」はありません．

何度も言うようですが，C言語のプログラミングの善し悪しを決める要素の1つは，これら構造体，共用体をいかにうまく使うかということなのです．

# 8.1　構造体の書式とその使い方

1つの char とか int などのデータ型は，コンピュータの内部構造に依存したデータ型であることは第4章で解説しました．この章で取り上げる構造体と共用体は，これらのデータ型を集めて，より複雑なデータをあたかも今までの変数と同じように扱えるように考えられたものです．

## ■構造体の考え方

複雑なデータとは，そのデータの大きさが任意のもので，またデータの種類(数値，文字列など)が単一でないデータであると考えることができます．たとえばよく引き合いに出されるものとして，以下の図8-1に示す「住所録」があります．これは，ある個人に関する各種のデータをひとまとめにしたものです．

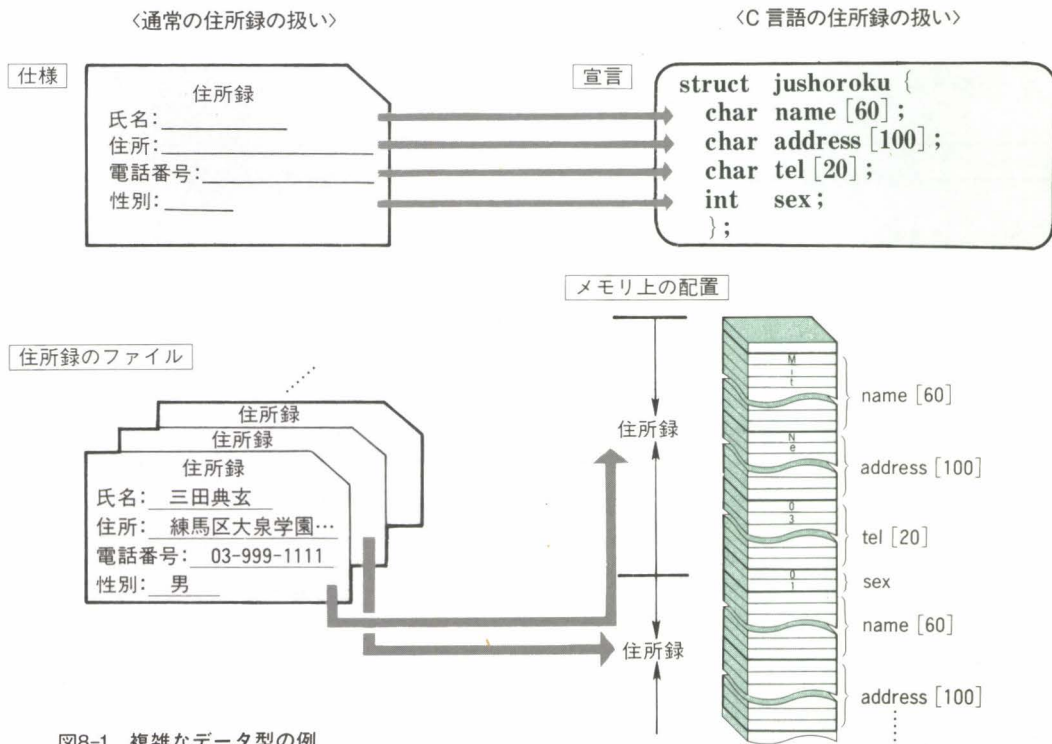



図8-1　複雑なデータ型の例

**図8-1**に示すように，C言語では住所録のようなデータの集まりを1つの単位として扱うことができます．これにより，各データのアクセスを非常に効率よく行うことが可能です．

## ■構造体の定義と宣言

それでは具体的に構造体の宣言方法について見てみましょう．まず，構造体(structure)の定義は以下の書式で行います．

```
struct　　構造体タグ{
    メンバー1の宣言；
    メンバー2の宣言；
        ⋮
    メンバーnの宣言；
};
```

**構造体タグ**は，一言でいえば新しいデータ型の名前であると考えることができます．そしてそのデータ型は，**メンバー**と呼ばれる1つ以上の要素から構成されます．この定義を先の**図8-1**と対応させてみましょう(**図8-2**)．

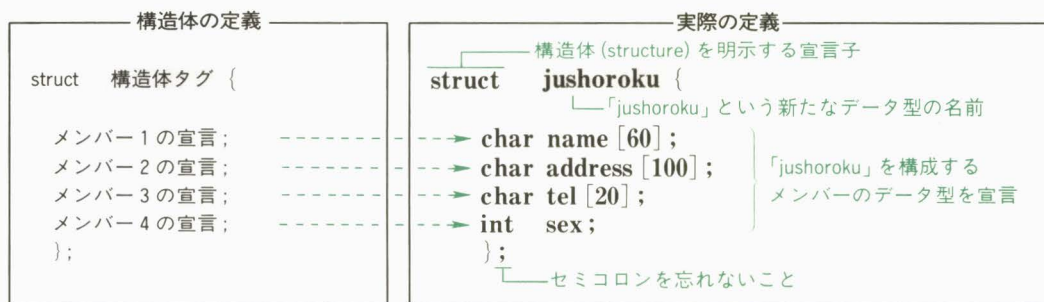



図8-2　構造体の定義

上記のような構造体の定義を行うと，intやcharなどと同じような新たなデータ型が作成されます．この定義でできる構造体は，データが入る容器を作り出すので**テンプレート**(型板)と呼ばれています．

ここではデータの型が定義されるだけなので，実際にはメモリ上にその領域が確保されるわけではありません．この構造体を使用するには，通常の変数と同じように以下のような使用宣言を行います．

```
struct　　構造体タグ　変数名[, 変数名…];
```

この宣言の書式は，たとえば次のような int 型の変数を宣言する場合とまったく同様です.

```
int   a,  b,  c;
```

この宣言によって構造体タグで示されるデータ型の変数が，以下の図8-3のようにメモリ上に確保されます.

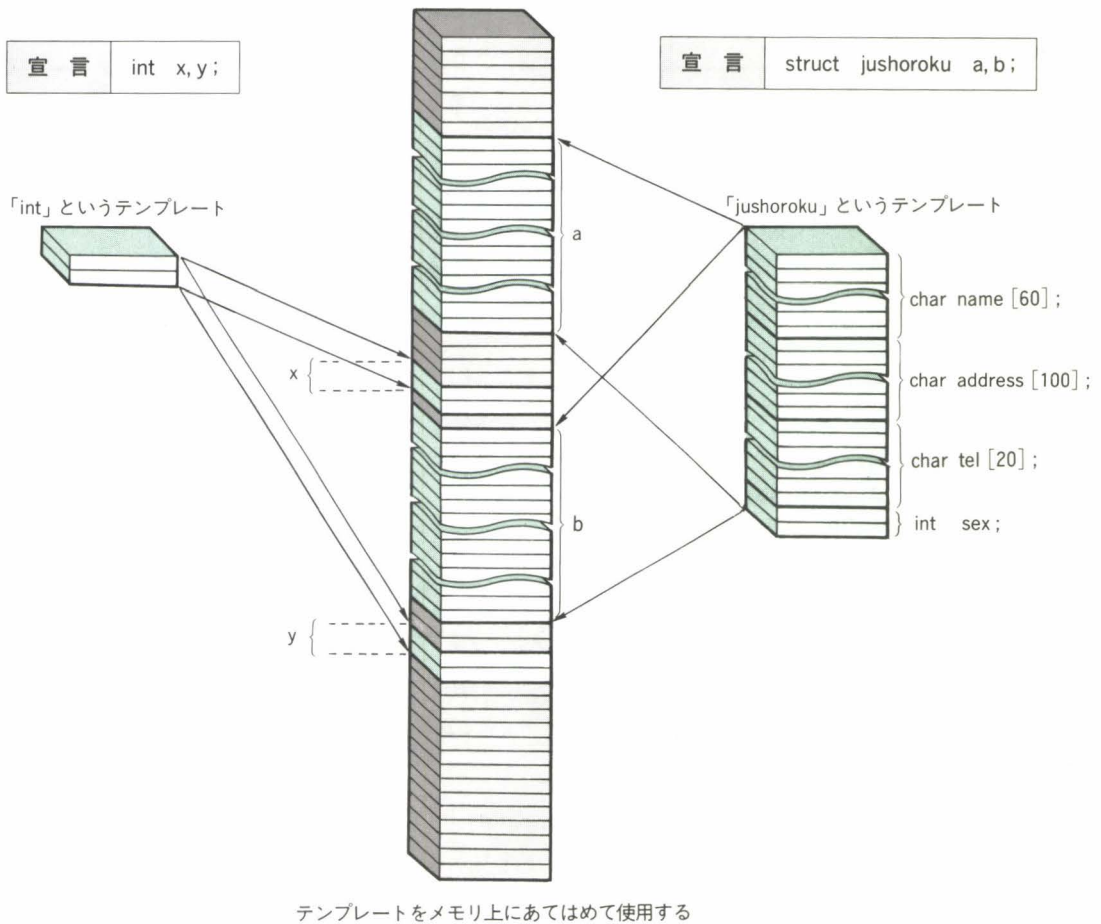



図8-3　構造体の使用宣言

構造体は下記のように，その定義と宣言を同時に行うこともできます．

```
struct    構造体タグ{
    メンバー1の宣言；
    メンバー2の宣言；
         ⋮
    メンバーnの宣言；
    }変数名[，変数名…]；
```

ここで構造体タグは，省略することが可能です．以下の**図8-4**に構造体の定義と宣言の例を示します．

```
struct    jushoroku {
    char  name[60];           ┐
    char  address[100];       │ 構造体の定義
    char  tel[20];            │
    int   sex;                ┘
    } a, b, c; ………構造体変数の宣言(定義のあとに変数名を並べる)

struct  jushoroku    x, y; ………構造体タグが定義してあるので，あとから別の変数も定義できる
```

```
struct    { ………構造体のタグ名を省略する
    char  name[60];           ┐
    char  address[100];       │ 構造体の定義
    char  tel[20];            │
    int   sex;                ┘
    } a, b, c; ………構造体変数の宣言

×struct  ?????????    x, y; ………構造体タグが定義されていないので，あとから別の変数を定義で
                                  きない
```

**図8-4　構造体の定義と宣言の例**

この節では，構造体を構成するメンバーとして，char, int などの基本データ型とその配列を用いていますが，実際にはポインタ変数，構造体自身なども宣言することができます．これらの例は以降で取り上げることにします．

## ■構造体の初期化

構造体は変数や配列と同様に，特定のデータであらかじめ初期化することができます．ただし初期化が可能なのは，

①記憶クラスとして「static」を指定する
②外部変数として宣言する

のいずれかの場合です．以下に構造体を初期化する書式を示します．

```
struct　構造体タグ　変数名＝{初期値[，初期値,……]};
```

構造体の初期化は，配列の場合と同じように考えればよいでしょう．また，次のように構造体の定義と同時に初期化を行うこともできます．

```
struct　　構造体タグ{
    メンバー1の宣言；
    メンバー2の宣言；
        ⋮
    メンバーnの宣言；
    }変数名＝{初期値[，初期値，……]}[，変数名＝……]；
```

以下の図8-5に構造体の初期化の例を示します．

```
main()
    {
    struct  jushoroku {
        char  *name;
        char  *address;     構造体の定義(ここでは，配列の代わりにポインタ変数を使っている)
        char  *tel;
        int    sex;
        }; ………セミコロンを忘れないこと

    static struct  jushoroku  a = {  ………static変数として構造体を初期化する
                                "Mita Norihiro",
                                "Nerima-ku Oizumigakuen",
                                "03-999-1111",
                                0     /* man */   配列の初期化と同じように各文字列
                                },                の先頭アドレスが構造体の各メンバ
                                                  ーにセットされる
```

次ページへ続く→

```
                    b = {
                        "Fujita Sachiyo",
                        "Adachi-ku Nisiarai",
                        "03-111-9999",
                        1     /* woman */
                        }; ………セミコロンを忘れないこと
        ⋮
    }
```

---

```
struct  jushoroku { ………外部変数として定義
    char  *name;          「jushoroku」というテンプレートは，以後使用しなければ省略することも可能
    char  *address;
    char  *tel;
    int    sex;
    } c = {      ………構造体変数の宣言と初期化を行う
          "Sato Eiichi",
          "Edogawa-ku Hirai",
          "03-555-5555",
          0     /* man */
          };

main()
    {
        ⋮
    }
```

図8-5　構造体の初期化の例

---

## ■構造体への代入と参照

構造体のそれぞれのメンバーは，構造体変数名とメンバー名を使って次の書式で参照できます．

| **構造体変数名 . メンバー名** |
|---|

ここで「.」(ピリオド)は，構造体変数とメンバーを結びつける**構造体メンバー演算子**です（43ページの「表2-8 演算子の優先順位表」を参照）．

先ほどの「住所録」を使った簡単なプログラムを組んで，構造体への代入や参照を行ってみることにしましょう(**リスト8-1**)．

```
 1: struct  jushoroku { ………構造体「jushoroku」の定義
 2:      char  *name;
 3:      char  *address;
 4:      char  *tel;
 5:      int    sex;
 6:      };
 7:
 8: main()
 9:      {
10:      struct  jushoroku    x, y, z; ………構造体変数x, y, z の宣言
11:
12:      x.name = "Mita Norihiro";    x.address = "Nerima-ku Oizumigakuen";
13:      x.tel = "03-999-1111";       x.sex = 0;
14:
15:      y.name = "Fujita Sachiyo";   y.address = "Adachi-ku Nisiarai";
16:      y.tel = "03-111-9999";       y.sex = 1;
17:
18:      z.name = "Sato Eiichi";      z.address = "Edogawa-ku Hirai";
19:      z.tel = "03-555-5555";       z.sex = 0;
20:
21:
22:      printf("%-14s -----> [Jusho]: %-25s¥n", x.name, x.address);
23:      printf("               -----> [tel]: %11s / [sex]: %d¥n", x.tel, x.sex);
24:
25:      printf("%-14s -----> [Jusho]: %-25s¥n", y.name, y.address);
26:      printf("               -----> [tel]: %11s / [sex]: %d¥n", y.tel, y.sex);
27:
28:      printf("%-14s -----> [Jusho]: %-25s¥n", z.name, z.address);
29:      printf("               -----> [tel]: %11s / [sex]: %d¥n", z.tel, z.sex);
30:      }
```

各構造体変数のメンバーに文字列や数値を代入

各構造体のメンバーを参照して，画面へ表示

[実行結果]

```
A>test
Mita Norihiro  -----> [Jusho]: Nerima-ku Oizumigakuen
               -----> [tel]: 03-999-1111 / [sex]: 0
Fujita Sachiyo -----> [Jusho]: Adachi-ku Nisiarai
               -----> [tel]: 03-111-9999 / [sex]: 1
Sato Eiichi    -----> [Jusho]: Edogawa-ku Hirai
               -----> [tel]: 03-555-5555 / [sex]: 0

A>
```

リスト8-1　構造体の参照

ここで，通常の変数と同様に構造体変数自身やメンバーのなかの要素は，次のように参照できます．

&x　…… 構造体変数の先頭アドレス

x.name[1]　…… x.name の2番目の文字

なお，構造体変数が「構造体へのポインタ変数」として宣言されている場合は，構造体メンバー演算子「－>」を使って各メンバーを参照します．これについては，次節でくわしく取り上げます．

# 8.2　構造体の利用

構造体といっても，その一般的な扱いは通常の char や int といった変数とあまり変わりませんから，記憶クラスもあれば配列もあります．また，ポインタ変数も扱うことができます．ここでは，そのような構造体の配列やポインタ変数について述べていくことにしましょう．

ただし処理系によっては，たとえば配列でも 2 次元配列は扱えないものがあるので注意してください．

## ■構造体の配列

前節の「住所録」では，それぞれ別々の構造体変数を用意しましたが，実際にはこれらの変数は，配列としてまとめておくと便利です．こうしておけば，住所録の各カードごとに，それぞれシリアルナンバーを付けるように管理することができます．

たとえば，先の住所録で 10 人分のデータを用意したければ，以下の**図 8-6** のように宣言することができます．

```
struct  jushoroku {
    char  *name;
    char  *address;
    char  *tel;
    int    sex;
    };

struct  jushoroku    data[10]; ………10 人分の住所録を配列として用意する
```

```
struct  jushoroku {  ………もちろん，構造体タグは省略してもよい
    char  *name;
    char  *address;
    char  *tel
    int    sex;
    } data[10]; ………構造体の定義と同時に配列の宣言を行う
```

図8-6　構造体の配列の宣言

ここで**図 8-6** のような宣言が行われた場合，以下のようにメモリが確保されます(**図 8-7**).

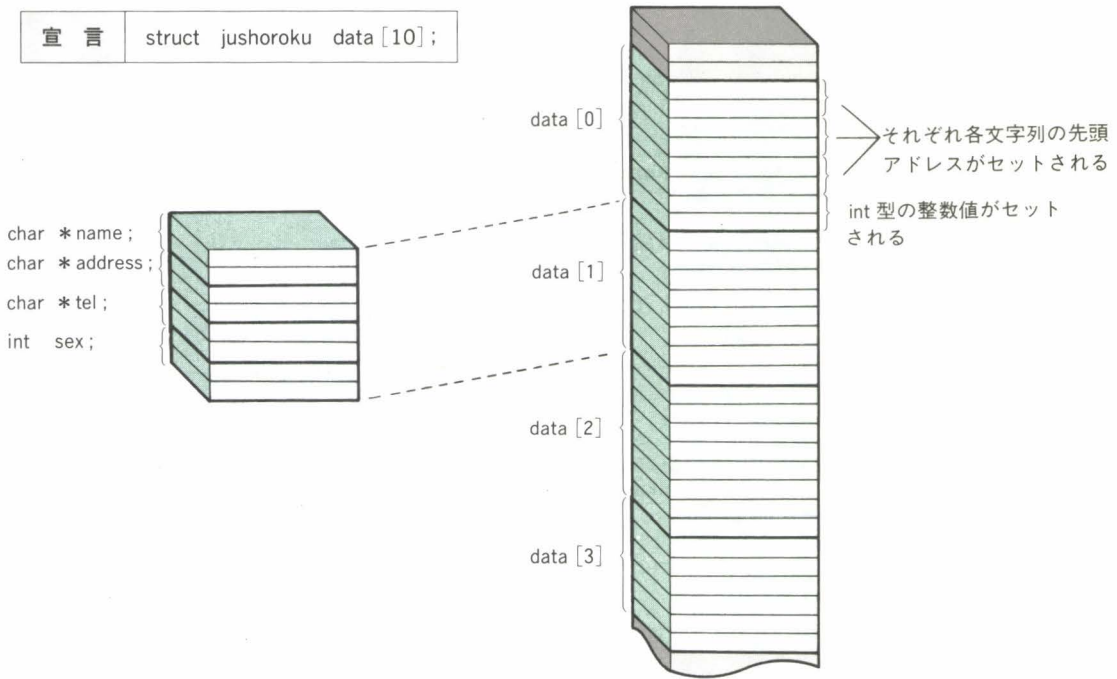



**図8-7　メモリの確保**

配列で宣言された構造体の各メンバーの参照は，先と同様に構造体メンバー演算子を用いて次のように行います.

**構造体配列名[要素番号].メンバー名**

前述の**リスト 8-1** のプログラムを構造体の配列を使って書き換えてみましょう.配列を使うことで，先の**リスト 8-1** と比べてこのような「住所録」の管理が簡単に行えるということがわかると思います.

次ページの**リスト 8-2** にそのプログラムを示します.

```
 1: struct  jushoroku {……構造体の定義
 2:     char  *name;
 3:     char  *address;
 4:     char  *tel;
 5:     int    sex;
 6:     };
 7:
 8: main()
 9:     {
10:     int     i;
11:     static struct  jushoroku    data[] = {……構造体の配列の宣言と初期化
12:         {"Mita Norihiro", "Nerima-ku Oizumigakuen",
13:          "Ø3-999-1111", Ø},
14:
15:         {"Fujita Sachiyo", "Adachi-ku Nisiarai",
16:          "Ø3-111-9999", 1}
17:         };
18:
19:     printf("No.    name             address                    tel       sex¥n");
20:     printf("-----------------------------------------------------------------¥n");
21:
22:     for(i = Ø ; i < 2 ; i++)
23:         {
24:         printf("%d  %-15s %-25s %-11s  %2d¥n",
25:                 i, data[i].name, data[i].address, data[i].tel, data[i].sex);
26:         }
27:
28:     printf("-----------------------------------------------------------------¥n");
29:     }
```

（注釈）data[]：配列の要素数は初期化する場合，省略することもできる
（注釈）25行目：配列を使うことで各構造体のメンバーの参照が簡単になる

［実行結果］

```
A>test↲
No.    name            address                    tel         sex
-----------------------------------------------------------------
Ø  Mita Norihiro   Nerima-ku Oizumigakuen     Ø3-999-1111    Ø
1  Fujita Sachiyo  Adachi-ku Nisiarai         Ø3-111-9999    1
-----------------------------------------------------------------

A>
```

リスト8-2　構造体の配列のサンプルプログラム

この例でもわかるように，「構造体の配列」といっても通常の変数の配列となんら変わるところはありません．

## ■構造体のポインタ変数

第5章のポインタ変数のところで解説したように，1次元配列とポインタ変数は同じように使用することができます．つまり，構造体のポインタ変数も，前節で取り上げた構造体の1次元配列と同じものです．

以下に「住所録」をポインタ変数として宣言した場合の宣言例とそのときのメモリの状態を示してみます(図8-8)．

| 宣 言 | struct jushoroku *data; |
|---|---|

〈ポインタ〉

-----宣言時に確保されるのは，この領域のみ

data →

data＋1 →

data＋2 →

*data

*(data＋1)

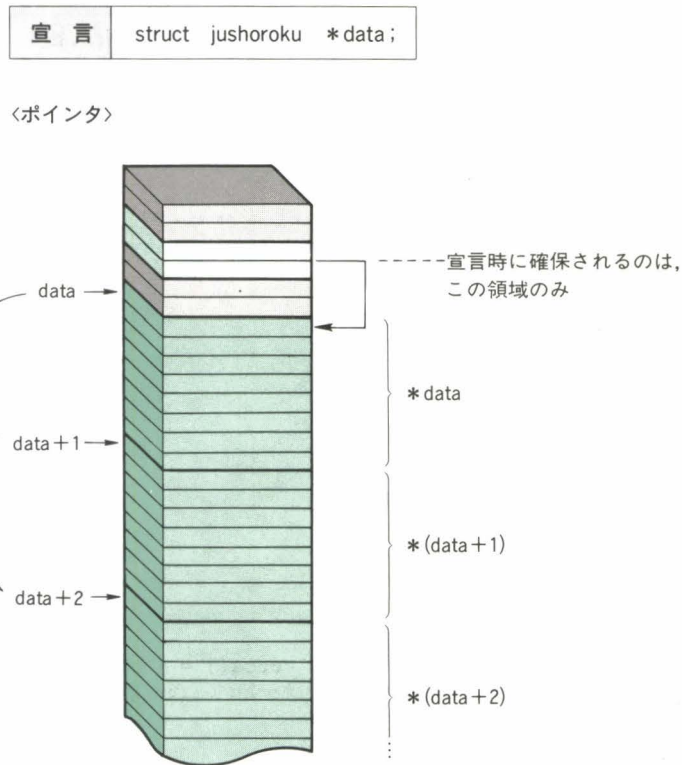
*(data＋2)

〈注意〉
ポインタ変数のポインタ変数(第5章参照)と同じように，この宣言では器だけが定義されその実体がないので，直接この変数を初期化したり，この変数に対して代入することはできない

図8-8 構造体のポインタ変数の宣言とメモリの状態

さて，ここでポインタ変数で宣言された構造体のメンバーを参照するにはどうしたらよいでしょうか？ 前節と同じ構造体メンバー演算子「.」を用いると次のように参照できます．

```
        ┌───────構造体のポインタ変数 data の n 番目のデータ
(＊(data＋n)). name  …… 優先順位に注意
               └構造体のメンバー名
```

しかし，このようにいくつもかっこを付けるのはわずらわしので，通常は次のような書式で参照を行います．

| 構造体のポインター＞メンバー名 |
|---|

この「－＞」は構造体のポインタ変数のメンバーを参照するときのみ使用できる構造体メンバー演算子です（43 ページの「表 2-8 演算子の優先順位表」を参照）．先の**リスト 8-2** を構造体のポインタ変数に変更して，実際に「－＞」演算子を使って見ることにしましょう（**リスト 8-3**）．

```
 1: struct  jushoroku {
 2:      char  *name;
 3:      char  *address;
 4:      char  *tel;
 5:      int    sex;
 6:      };
 7:
 8: main()
 9:      {
10:      int      i;
11:
12:      struct  jushoroku    *s_ptr; ………構造体のポインタ変数の宣言
13:
14:      static struct  jushoroku    data[] = {
15:           {"Mita Norihiro", "Nerima-ku Oizumigakuen",
16:            "03-999-1111", 0},
17:
18:           {"Fujita Sachiyo", "Adachi-ku Nisiarai",
19:            "03-111-9999", 1}
20:           };
21:
22:      printf("No.    name               address                   tel       sex¥n");
23:      printf("----------------------------------------------------------------¥n");
24:
25:      s_ptr = data;………構造体のポインタ変数は初期化や代入ができないので，構造体の配列として
                                                初期化した data[ ]の先頭アドレスを受け取る
26:      for(i = 0 ; i < 2 ; i++, s_ptr++)
27:           {              └────── ポインタ変数なのでインクリメントが可能
```

```
28:         printf("%d  %-15s %-25s %-11s  %2d¥n",
29:                 i, s_ptr->name, s_ptr->address, s_ptr->tel, s_ptr->sex);
30:         }
31:
32:     printf("-----------------------------------------------------------¥n");
33:     }
```

└構造体メンバー演算子「－＞」を使った構造体のメンバーの参照

［実行結果］

```
A>test
No.   name              address                  tel         sex
------------------------------------------------------------------
0  Mita Norihiro   Nerima-ku Oizumigakuen     03-999-1111    0
1  Fujita Sachiyo  Adachi-ku Nisiarai         03-111-9999    1
------------------------------------------------------------------

A>
```

リスト8-3　構造体のポインタ変数の使用例

演算子の優先順位表を見ればわかるように，構造体メンバー演算子である「.」と「－＞」は非常に高い優先順位をもっています．たとえば，

＋＋data－＞name

という演算では「－＞」が優先されるので，あえてかっこを付けて表現すると，

＋＋(data－＞name)

となります．つまり，「ポインタで表される構造体 data のメンバーname を１つインクリメントする」という意味になります．したがって，「data＋１の構造体のメンバーname を参照する」場合には以下のようにかっこを付けて表現する必要があります．

(＋＋data)－＞name

## ■構造体の受渡し

前節で示したように，構造体のポインタ変数は直接初期化したり，代入を行うことはできません．実際には，構造体のポインタ変数は関数へ構造体を受け渡す際によく用いられます．というのは，構造体の受渡しには次のような制約があるからです．

① 構造体を直接関数に渡したり，返したりすることはできない
② 構造体のデータを一度に同じデータ型の構造体に代入することはできない

つまり構造体の受渡しや代入には，構造体のポインタやメンバーそれぞれのアドレスを使わなければならないのです(ただし，最近のパソコン上のC言語ではこのような制約がない処理系も増えてきている)．そこでここでは，構造体のポインタ変数を用いた関数との受渡しの例題を見てみることにしましょう．以下に，構造体のポインタを関数に受け渡すプログラムを示します(**リスト 8-4**)．

```
 1: #include     <stdlib.h> ………atoi 関数が宣言されているヘッダファイル
 2: #include     <stdio.h>
 3:
 4: struct  jushoroku {
 5:         char   name[30];
 6:         char   address[50];
 7:         char   tel[20];
 8:         int    sex;
 9:         };
10:
11: typedef  struct jushoroku    MEIBO; ………構造体 jushoroku を新たなデータ型として定義
12:
13: int     input(i_data) …入力用の関数
14: MEIBO     *i_data; ………構造体のポインタが main 関数から渡される
15:         {
16:       char     buf[20];          /* temp. buff */
17:       char     *gets(); ………関数の使用宣言．stdio.h のなかであらかじめ宣言されていれば必要ない
18:
19:         printf("name? : ");                    gets(i_data->name); ……構造体へ文字列を入力する
20:         if(i_data->name[0] == 0)               return(0); ………1 文字目がヌル文字なら入力終了
21:
22:         printf("address? : ");                 gets(i_data->address);
23:
24:         printf("tel? : ");                     gets(i_data->tel);
25:
26:         printf("sex(0=man/1=woman)? : ");      i_data->sex = atoi(gets(buf));
27:
28:         printf("¥n");
29:         return(1);
30:         }
31:
32: void     display(d_data, di) ………表示用の関数
33: MEIBO     *d_data; ………構造体のポインタが main 関数から渡される
34: int        di;
35:         {
36:         int     j;
37:
38:         printf("No.   name              address                    tel      sex¥n");
39:         printf("-----------------------------------------------------------------¥n");
40:
41:         for(j = 0 ; j < di ; j++, d_data++)
42:             {
43:             printf("%d  %-15s %-25s %-11s  %2d¥n",
44:                     j + 1, d_data->name, d_data->address, d_data->tel, d_data->sex);
45:             }
```

```
46:
47:     printf("-----------------------------------------------------------¥n");
48:     }
49:
50: main()
51:     {
52:     MEIBO    data[10];
53:     MEIBO    *s_ptr = data; ………構造体のポインタを宣言し，構造体の実体（data[10]）のアドレスを代入する
54:
55:     int      i;
56:
57:     for(i = 0 ; i < 10 ; i++, s_ptr++)
58:         if(!input(s_ptr))    break;
59:
60:     printf("¥n¥n");
61:     display(data, i);
62:     }
```

［実行結果］

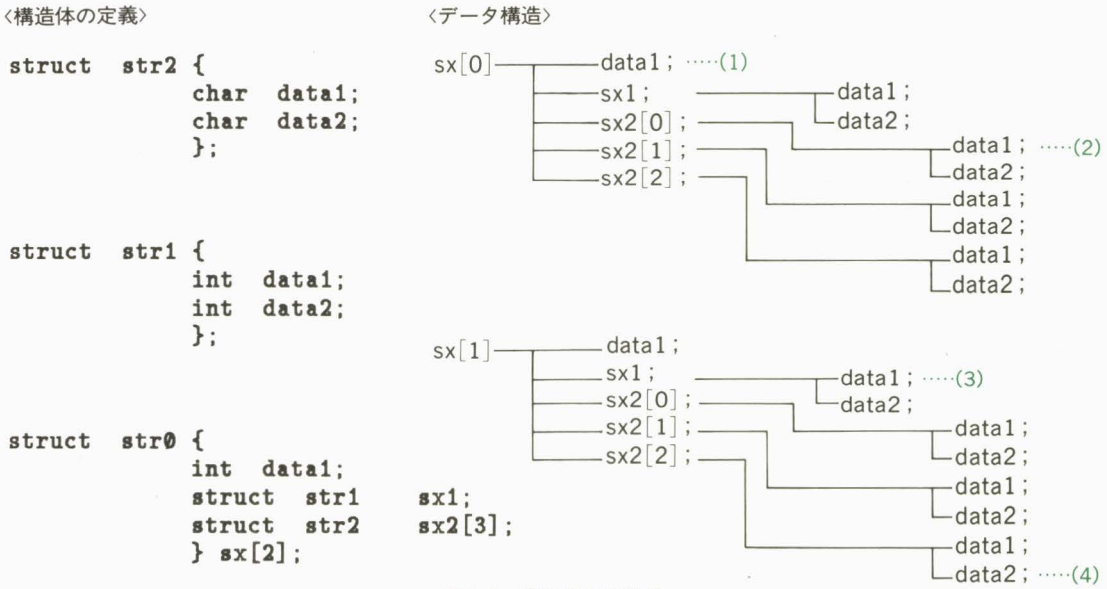
```
A>test
name? : Mita Norihiro
address? : Nerima-ku Oizumigakuen
tel? : 03-999-1111
sex(0=man/1=woman)? : 0

name? : Fujita Sachiyo
address? : Adachi-ku Nisiarai
tel? : 03-111-9999
sex(0=man/1=woman)? : 1

name? :  ………リターンで入力終了


No.    name              address                  tel         sex
------------------------------------------------------------------
0  Mita Norihiro   Nerima-ku Oizumigakuen    03-999-1111   0
1  Fujita Sachiyo  Adachi-ku Nisiarai        03-111-9999   1
------------------------------------------------------------------

A>
```

リスト8-4　構造体のポインタを関数に受渡す例

多くの場合，構造体を使ってこのような住所録を管理するには，**リスト 8-4** のように配列を使ってデータ量を固定してしまうのではなく，構造体のチェインという手法を用いてデータ量に応じたダイナミックなメモリ管理を行います．これらの具体的な例は「8.4 自己参照構造体」のところで取り上げることにします．

## ■構造体の構造体

構造体のメンバーには，構造体自身を記述することができます．これはC言語で構造体を使う上で最も便利な点であり，また実際によく利用されます．構造体の構造体を使うと，たとえば以下の**図8-9**に示すような階層化されたデータを簡潔に記述することが可能です．

図8-9　構造体の構造体

ここで，各構造体のメンバーを参照するには，これまでと同様に構造体メンバー演算子「.」，「－＞」を用いて行います．階層の下の方のデータは，構造体のメンバーを演算子でつなげていくことで，その実体にたどりつくことができます．**表8-1**に先の**図8-9**の各メンバーの参照例を示します．

| 宣言 | (1)のデータ | (2)のデータ | (3)のデータ | (4)のデータ |
|---|---|---|---|---|
| 通常の変数 | sx[0].data1 | sx[0].sx2[0].data1 | sx[1].sx1.data1 | sx[1].sx2[2].data2 |
| ポインタ変数 | (sx+0)－＞data1 | (sx+0)－＞(sx2+0)－＞data1 | (sx+1)－＞sx1－＞data2 | (sx+1)－＞(sx2+2)－＞data2 |

表8-1　構造体の構造体のメンバーの参照

**図8-9**では，構造体のなかに別の構造体が含まれていましたが，よく使われるのは構造体のなかに自分自身の構造体を含む場合です．このような構造体を作ることで，チェインやツリー構造のデータの管理を行うことができます．

# 8.3 FILE構造体

第7章で取り上げた高水準入出力関数は，ファイルの情報を定義した構造体として「stdio.h」ヘッダファイルのなかのFILE構造体を使用しています．このFILE構造体は，構造体の仕組みを理解する上でちょうどよいサンプルです．ここでは，FILE構造体の中身を調べてみることによって，実際にどのように構造体が使われているのかを見てみましょう．

## ■FILE構造体の中身

まずは，「stdio.h」ヘッダファイルのなかのFILE構造体の定義とその構造体が使われている部分を抜き出してみます(図8-10)．

```
 1: #define  FILE    struct _iobuf
 2:
 3: extern FILE {
 4:     char *_ptr;
 5:     int   _cnt;
 6:     char *_base;
 7:     char  _flag;
 8:     char  _file;
 9:     } _iob[_NFILE];
10:
11: #define  stdin   (&_iob[0])
12: #define  stdout  (&_iob[1])
13: #define  stderr  (&_iob[2])
14: #define  stdaux  (&_iob[3])
15: #define  stdprn  (&_iob[4])
16:
17: #define  getc(f)    (--(f)->_cnt >= 0 ? 0xff & *(f)->_ptr++ : _filbuf(f))
18: #define  putc(c,f)  (--(f)->_cnt >= 0 ? 0xff & (*(f)->_ptr++ = (c)) : ¥
19:                 _flsbuf((c),(f)))
20:
21: #define  getchar()   getc(stdin)
22: #define  putchar(c)  putc((c),stdout)
23:
24: #define  feof(f)     ((f)->_flag & _IOEOF)
25: #define  ferror(f)   ((f)->_flag & _IOERR)
26: #define  fileno(f)   ((f)->_file)
```

〈注意〉
- この図版はMicrosoft C Compilerの「stdio.h」ファイルから抜粋して作成した

図8-10　FILE構造体の中身

1行目では #define 文によって，FILE という構造体の名前は実は「_iobuf」という名前の構造体であることが示されています．また，3行目からの構造体の定義の部分では，構造体そのものがこの「stdio.h」のなかで作られているわけではなく，どこか別のコンパイル単位で作られたものを extern で持ってきて使っているということがわかります．

FILE 構造体のメンバーは，以下のような要素から構成されています．これは Microsoft C Compiler の場合です．他の処理系でも定義の仕方は異なりますが，ほぼ同じ内容が定義されています(各処理系の FILE 構造体の中身については APPENDIX を参照のこと)．

```
char   *_ptr;   …… 現在ファイルを読み書きしている位置(ファイルポインタ)
int     _cnt;   …… ファイルポインタからファイルの最後までのデータ数
char   *_base;  …… データを読み書きするバッファの位置
char    _flag;  …… ファイルにエラーが起こったかどうかを表すフラグ
char    _file;  …… 現在のファイルが何番目にオープンされたファイルかという情報
```

ここで示すように5つの変数が集まって FILE 構造体が構成されています．そして，この構造体のメンバーは，ファイルのオープンやクローズ，ファイルのリード／ライトなどファイル操作を行うときに参照されます．

また9行目を見ると，この FILE 構造体の実体が「_iob」という配列で定義されていることがわかります．FILE 構造体は，オープンするファイルにかならず1つずつ割り当てられますから，この配列の大きさがそのC言語で一度に扱えるファイルの数を決定します(ただし標準入出力など，あらかじめオープンされるファイルもその数に含まれているという点に注意してください)．

## ■FILE 構造体の利用

前節で定義された FILE 構造体が実際に，どのように利用されているのかを見ていきましょう．もう一度図 8-10 を見てください．11 行目に次のような定義があります．

```
#define   stdin   (&_iob[0])
```

「stdin」は，第7章で学んだように標準入力(通常はキーボード)のFILE構造体へのポインタです．ここで「stdin」は「&_iob[0]」として定義されており，FILE構造体の1番目の配列の先頭アドレス(ポインタ)を指していることがわかります．以下同じように標準出力や標準エラー出力なども，あらかじめFILE構造体へのポインタが割り当てられているのです．

次に 18 行目を見ると，putc関数がマクロで定義してあります．これを解析してみましょう．かなり複雑なマクロ定義ですが，なかをじっくり見てみると，構造体のメンバーの参照や三項演算子が使われているのがわかります．以下の図8-11にputc関数の解析結果を示します．

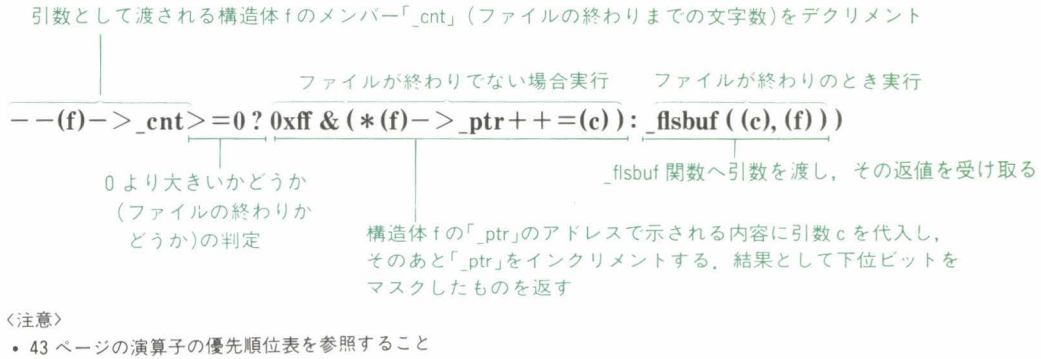



図8-11 putc関数の解析

その他のマクロ定義された関数についても，同じように解析してみてください．実際のプログラムでは，FILE構造体を宣言すると，メンバーそのものを参照しなければならない場面はめったにありません．しかし，リードやライトなどの関数のなかでは実際にはこのように使われているのです．ちょっと細かいファイルの制御を行う場合は，このFILE構造体のメンバーを直接参照することもありますから覚えておくとよいでしょう．

## ■FILE構造体の中身を調べるプログラム

C言語のプログラムでは，よほど大きなものを組まない限り自分で構造体を作るよりも，システムや処理系で用意された構造体を使うことの方が圧倒的に多いといえるでしょう．これまで解説してきたFILE構造体などはそのよい例であり，このほかにもディスクのディレクトリの構造体などもよく使われます．

そこで，これらのすでに定義してある構造体の中身を調べるプログラムを作ってみましょう．ここでは，Microsoft C CompilerのFILE構造体を例とします．この構造体はオペレーティング・システムや処理系に大きく依存しますが，どの処理系でも似たり寄ったりですから，他の処理系をお持ちの方は「stdio.h」をよく読んで同じことをしてみてください．

以下にプログラムリストとその実行結果を示します(**リスト8-5**)．

```
 1: /*
 2:     Read and Display FILE-Structure
 3:     1986/07/24 by N.Mita
 4: */
 5:
 6: #include    <stdio.h>
 7:
 8: main(argc, argv)
 9: int         argc;
10: char     **argv;
11:     {
12:     FILE     *fp;  ………FILE 構造体へのポインタ変数を宣言
13:     char     buffer[20];
14:     int      i;
15:     int      c;
16:
17:     if(argc != 2)    exit(0); ………ファイル名を入れ忘れたら何もせずに抜ける
18:
19:     if(NULL == (fp = fopen(argv[1], "r"))) ………指定ファイルをオープン
20:         {          └── 割り当てられたファイル構造体のポインタが返る
21:         puts("¥7FILE-OPEN ERROR.¥n");
22:         exit(-1);
23:         }
24:
25:     for(i = 0 ; i < 0x7FFF ; ++i)
26:         {
27:         gets(buffer); ………入力があるまで待つ
28:         if((buffer[0] == 'e') || (buffer [0] == 'E'))    break;……… e または E で
29:         c = fgetc(fp); ………ファイルから 1 文字入力                      プログラム終了
30:         printf(" No. %08d¥n", i);
31:         printf(" ===> character data is     :  %c¥n", c);
32:         printf(" ===>  _ptr  is             :  %08lX¥n", fp->_ptr);
33:         printf(" ===> *_ptr  is             :  %02X¥n", *fp->_ptr);
34:         printf(" ===>  _cnt  is             :  %04X¥n",  fp->_cnt);
35:         printf(" ===>  _base is             :  %08lX¥n", fp->_base);
36:         printf(" ===> *_base is             :  %02X¥n", *fp->_base);
37:         printf(" ===> _flag  is             :  %02X¥n",  fp->_flag);
38:         printf(" ===> _file  is             :  %02X¥n¥n",fp->_file);
39:         }                                          └ FILE 構造体のメンバーを表示
40:
41:     fclose(fp); ………ファイルのクローズ
42:     }
```

［実行結果］

```
A>test↵
   ↵      ………何かキーを入力する
 No. 00000000
 ===> character data is      :  /          ………………1 文字目のデータを表示
 ===>  _ptr  is              :  02140FD7
 ===> *_ptr  is              :  2A         ………………1 文字目のキャラクタコード
```

```
===>  _cnt  is            :  01E4
===>  _base is            :  02140FD6 ………………バッファのアドレス
===> *_base is            :  2F
===> _flag  is            :  09
===> _file  is            :  05  ………このファイルは6番目にオープンされたファイルである
  ↵                                （実際には、標準入出力としてあらかじめ5つのファイルがオープン
No. 00000001                       されているので、プログラム中では初めてオープンされたファイル）
===> character data is    :  *
===>  _ptr  is            :  02140FD8
===> *_ptr  is            :  0A
===>  _cnt  is            :  01E3 ………1文字読むごとに残りの文字数は1つずつ小さくなる
===>  _base is            :  02140FD6
===> *_base is            :  2F
===> _flag  is            :  09
===> _file  is            :  05
  ↵
No. 00000002
===> character data is    :

===>  _ptr  is            :  02140FD9 ………1文字読むごとにポインタはインクリメントされる
===> *_ptr  is            :  20
===>  _cnt  is            :  01E2
===>  _base is            :  02140FD6
===> *_base is            :  2F
===> _flag  is            :  09
===> _file  is            :  05
  ↵
No. 00000003
===> character data is    :
===>  _ptr  is            :  02140FDA
===> *_ptr  is            :  20
===>  _cnt  is            :  01E1
===>  _base is            :  02140FD6
===> *_base is            :  2F
===> _flag  is            :  09
===> _file  is            :  05
  ↵
No. 00000004
===> character data is    :
===>  _ptr  is            :  02140FDB
===> *_ptr  is            :  20
===>  _cnt  is            :  01E0
===>  _base is            :  02140FD6
===> *_base is            :  2F
===> _flag  is            :  09
===> _file  is            :  05

e ↵ ……… e または E で表示終了

A>
```

リスト8-5 FILE構造体の中身を調べるプログラム

# 8.4 複雑な構造体

この節では構造体を使った応用例として，自己参照構造体とビットフィールドについて取り上げます．とくに自己参照構造体は，リスト構造でデータを管理する際の要となるものですから，よく理解してください．なお，ビットフィールドは処理系によってサポートしていないものもあるので注意が必要です．

## ■ 自己参照構造体

構造体のメンバーとして構造体を含むことができるということは，前の節で解説しました．ところで，この構造体のネストのなかには，自分自身の構造体を入れてしまうことも可能なのです．なんだかややこしく感じるかもしれませんが，プログラムでいえば「再帰」のようなデータ構造を作ることができるのです．

ともあれ，先の「住所録」を使って自己参照構造体を作ってみることにしましょう(図8-12)．

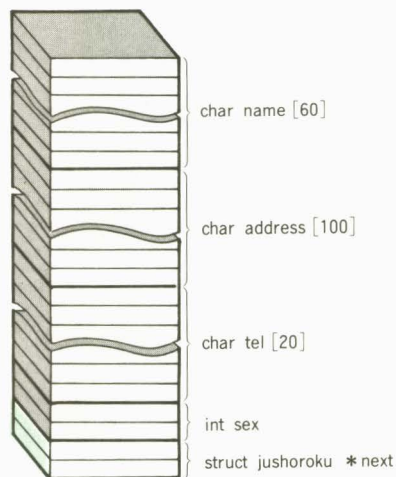



図8-12　自己参照構造体の宣言例

ここでは,「jushoroku」という構造体のメンバーのなかに「struct　jushoroku　＊next;」という自分自身の構造体へのポインタが含まれています．これは，jushorokuで宣言される構造体変数へのポインタという意味になります．つまり以下の図8-13で示すように,「next」に構造体変数のポインタを代入していくことで，構造体を次々につなげていく(**チェインする**)ことができるわけです．

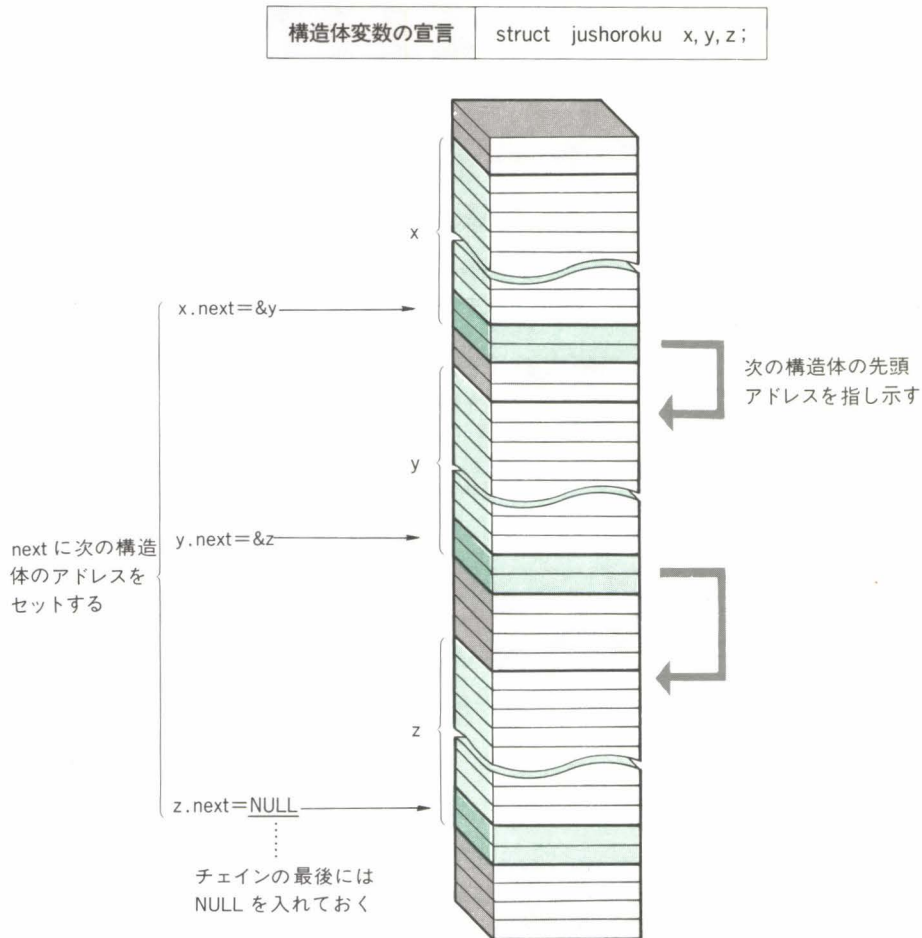



図8-13　構造体のチェイン

この構造体を参照するには，次のように最初の構造体のアドレスをポインタ変数にセットしておくと，

```
struct　jushoroku　*ptr;
ptr　=　&x;
```

その次の構造体 y は「ptr－＞next」というかたちで参照することができます．以降の構造体もポインタをたどっていくことで同じように次々に参照することが可能です．

このようなデータ構造を**線形リスト**と呼びますが，これは同じようにメモリ上に領域を確保する配列とは異なり以下のような特徴をもっています．

① データ数が固定していないものを効率よく扱える
② 挿入や削除が物理的な移動を伴わずに行える

配列では，あらかじめ固定的にメモリ上にその領域が確保されてしまいますが，線形リストでは，たとえばmalloc関数やfree関数と組み合わせることで必要に応じてメモリ領域を確保することができます．また，線形リストのデータはそれぞれポインタのチェインによって結ばれていますから，そのチェインをつなぎかえることによって，データの挿入や削除が簡単に行えます(**図8-14**)．

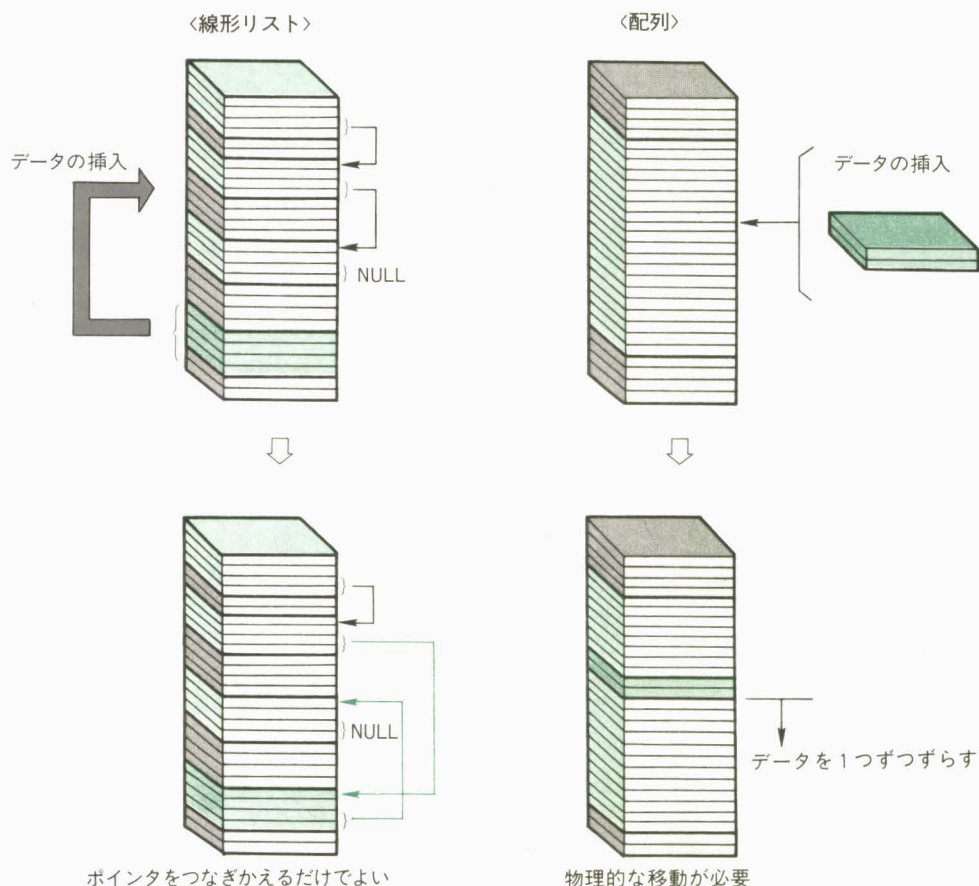



図8-14　線形リストと配列

しかし，線形リストでは特定のデータを検索する際にかならずチェインの先頭から1つずつポインタをたどっていかなければならないという欠点があります．とくにデータ量が増えた場合は，その検索効率が悪化してしまいます．

つまり，線形リストはダイナミック(動的)にデータが変化するものに対して有効であり，配列は固定的なデータに対して有効です．たとえば線形リストは，エディタやLISPの記述などに使われます．C言語は，構造体とポインタ変数を持っているおかげで，このような動的なデータ管理を行う線形リストが非常に書きやすい構造となっています．

## ■ 自己参照構造体のサンプルプログラム

自己参照構造体を使ったプログラムを実際に組んでみることにしましょう．また，第10章で紹介する「xref」(クロスリファレンサ)もこの構造体のリンクを十分に活用したプログラムですので参照してください．

ここでは，簡単なゲームを作ってみました．このゲームでは，あらかじめ問題をいくつか作成し，あとからそれに答えていくという方式をとっています．ただし，作ったデータをディスクにしまい込む機能は用意していません．したがって，このプログラムではゲームが終った段階で，すべてのデータが消えてしまいます．やる気のある方は拡張を試みてください．

このプログラムの特徴は，問題の個数によってその都度必要なメモリを確保していくということです．1ゲームのデータはすべて1つの構造体のなかに入っており，またその構造体には前ゲームのデータが入っている構造体のポインタと次ゲームのデータが入っている構造体のポインタも含まれています．つまり，データのチェインができているので，ゲームの進行を自在に行うことができます．

このゲームのプログラムとその遊び方を以下に示します(**リスト8-6**)．

```
 1: /*
 2:
 3: Word Response Games
 4:
 5: */
 6:
 7:
 8: #include    <stdio.h>
 9: #include    <ctype.h>
10:
11: #ifdef  MSDOS
12: #include    <malloc.h>  ………MS-DOS 上のC 言語では malloc.h をインクルードする
13: #endif  /* MSDOS */
14:
15: #define     GOOD    (-1)
16: #define     POOR    0
```

```
17:                                    ┌── typedef されているため，構造体タグ名 talk は，
18:                                    │   1つのデータ型と見なされる
19: typedef struct  talk    {
20:         struct  talk    *previous; … 前のデータが入っている構造体へのポインタ
21:         struct  talk    *forward;  … 次のデータが入っている構造体へのポインタ
22:         char    command[80];  ………… 問題を入れる文字バッファ
23:         char    response[80]; ………… 解答を入れる文字バッファ
24:         int     timeout;      ………… 失敗した場合のリトライの回数
25:         } talkv, *talkp;
26:               └── 構造体の実体
27:
28: struct  talk    *cp; ……… ゲームで使う最初の構造体を指すポインタの宣言
29:
30:
31: main()                                        構造体のバイト数
32:     {
33:     if((talkp)NULL == (cp = (talkp)malloc(sizeof(talkv))))
34:         {
35:         printf("¥7***** No memories .... Sorry ....¥n"); 最初の構造体を入れるエリアを
36:         exit(0);                                         確保する
37:         }
38:
39:     mkgame(cp); ……… 問題の作成
40:
41:     if(GOOD == game(cp)) ……… ゲームの実行と終了の判断
42:         printf("You are winner !¥7¥n");
43:     else
44:         printf("Poor! Retry more!");
45:     }
46:
47:
48: mkgame(c)
49: struct  talk    *c; ……… 最初の構造体のポインタが渡される
50:     {
51:     char    c_buff[80];
52:     char    r_buff[80];
53:     char    t_buff[80];
54:     struct  talk    *curr, *xcp;
55:
56:     curr = (talkp)NULL; ……… ダミーの構造体を NULL でセット
57:
58:     for(;;)
59:         {
60:         xcp = curr; } 1つ前のポインタを保存する
61:         curr = c;   }
62:
63:         if((talkp)NULL == (c = (talkp)malloc(sizeof(talkv)))) ……… 次の構造体を入れる
64:             {                                                       エリアを確保する
65:             printf("¥7***** No memories .... Sorry ....¥n");
66:             exit(0);
67:             }
68:
69:         curr->previous = xcp; } ポインタのチェインを作る
70:         curr->forward  = c;   }
```

<構造体 talk>

```
 71:
 72:         printf("Enter talk of Computer : ");  ………問題の入力
 73:         gets(c_buff);
 74:
 75:         if(0 == strncmp("END", c_buff, 3))  ………入力が END ならば問題の作成を終了する
 76:             {
 77:             curr->forward = (talkp)NULL;  ………最後の構造体の curr－＞forward には
 78:             break;                                 かならず NULL をセットしておく
 79:             }
 80:
 81:         printf("Enter Responses of man : ");  ………解答の入力
 82:         gets(r_buff);
 83:
 84:         printf("Enter Max of response  : ");  ………リトライできる回数の入力
 85:         gets(t_buff);
 86:
 87:         puts("¥n");  ………2 行送る
 88:
 89:         strncpy(curr->command,  c_buff, 80);  ┐
 90:         strncpy(curr->response, r_buff, 80);  │ 入力されたデータを構造体のメンバーにセット
 91:         (*curr).timeout = atoi(t_buff);       ┘
 92:         }
 93:     }
 94:
 95:
 96: int game(ptr)  ………ゲームを実行する関数
 97: struct  talk    *ptr;  ………最初の構造体のポインタが渡される
 98:     {
 99:     for(;;)                                       ………次の構造体を指し示すポインタが
100:         {                                             NULL となるまでゲームを操り返す
101:         if((talkp)NULL == ptr->forward)    break;
102:         if(GOOD != game_1(ptr))            return(POOR);  ………問題が解けなければ
103:         else                               ptr = ptr->forward;  「POOR」を返す
104:         }
105:     return(GOOD);                          次の構造体のポインタをセットする
106:     }
107:
108:
109: int game_1(curr)  ………1 ゲームを実行するプログラム
110: struct  talk    *curr;  ………構造体のポインタが渡される
111:     {
112:     int i;
113:     char    buffer[80];
114:
115:     printf("¥n¥nI talk %s.¥n¥n>>>>>>> Response ? ¥n",curr->command);  ………問題を
116:                                                                              表示
117:     for(i = 0 ; i < curr->timeout ; ++i)  ………リトライの回数だけ繰り返す
118:         {
119:         printf("%3d : ", i+1);
120:         gets(buffer);  ………解答の入力
121:         if(0 == strcmp(curr->response, buffer))  ………正解との比較
122:             {
123:             printf("Good !¥n");
124:             return(GOOD);
```

```
125:                }
126:            }
127:        return(POOR);
128:        }
```

［実行結果］

```
A>GAME ↵        ………ゲームの実行．まず問題を作成する

Enter talk of Computer  : What's the biggest city in Japan?↵   ………問題
Enter Responses of man  : Tokyo ↵   …………解答
Enter Max of response   : 4 ↵       …………リトライの回数


Enter talk of Computer  : 2 + 3 = ?↵
Enter Responses of man  : 5 ↵
Enter Max of response   : 5 ↵


Enter talk of Computer  : Which is the bell code in ASCII?↵
Enter Responses of man  : 7 ↵
Enter Max of response   : 4 ↵


Enter talk of Computer  : END ↵   ………問題の作成を終了する
```

〈Q & A のスタート〉

```
I talk What's the biggest city in Japan?.

>>>>>>> Response ?
  1 : Osaka ↵
  2 : Tokyo ↵
Good !
I talk 2 + 3 = ?.

>>>>>>> Response ?
  1 : 4↵
  2 : 5↵
Good !


I talk Which is the bell code in ASCII?.

>>>>>>> Response ?
  1 : 7↵
Good !
You are winner !

A>
```

リスト8-6　ゲーム・プログラム

このプログラムで使われている構造体のつながり(チェイン)を以下の**図8-15**に示しておきます．このようなデータ構造は，２つのリンクポインタをもっているので，ダブルリンク構造と呼ばれます．

実際にはこのプログラムでは，ゲームの進行が１方向なので「previos」ポインタを活用していませんが，解答に応じて問題を選択するようなインタラクティブな構成にするとこの２つのポインタが生きてきます．みなさんで改良してみてください．

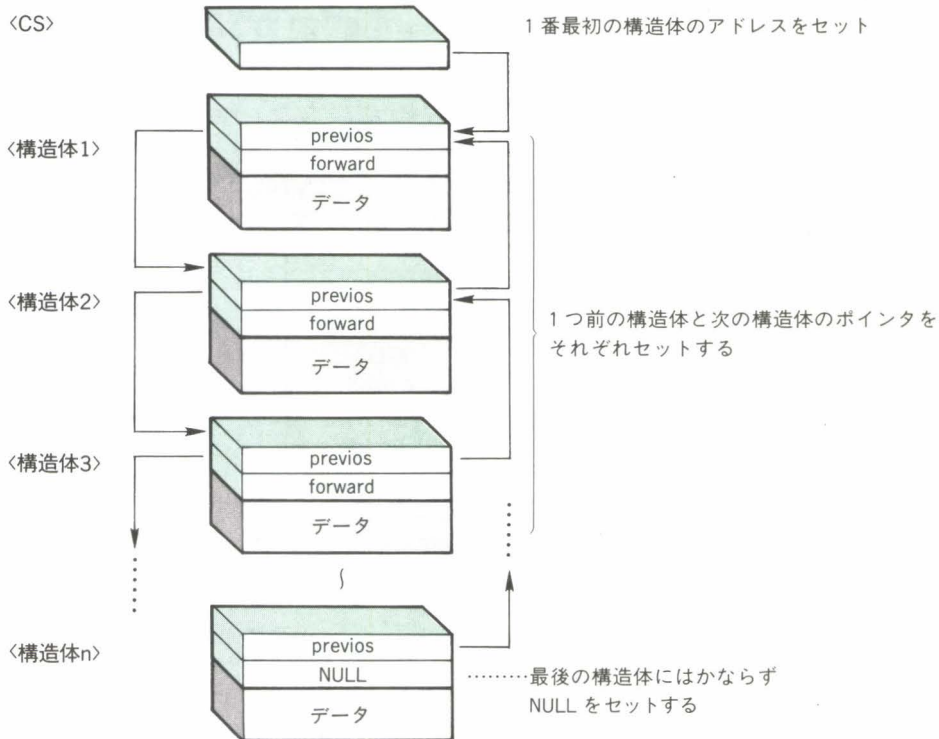



**図8-15 ダブルリンク構造**

## ■ビットフィールド

構造体の便利な機能として，ビット単位でデータを扱えるビットフィールドの機能があります．このビットフィールドを用いると，たとえば周辺機器などの制御で使われるフラグの集合を簡単に取り扱うことができます．

それでは具体的に，ビットフィールドを定義してみましょう．以下の図8-16は，RS-232Cを使ってデータ通信を行うためのビットの集合です．

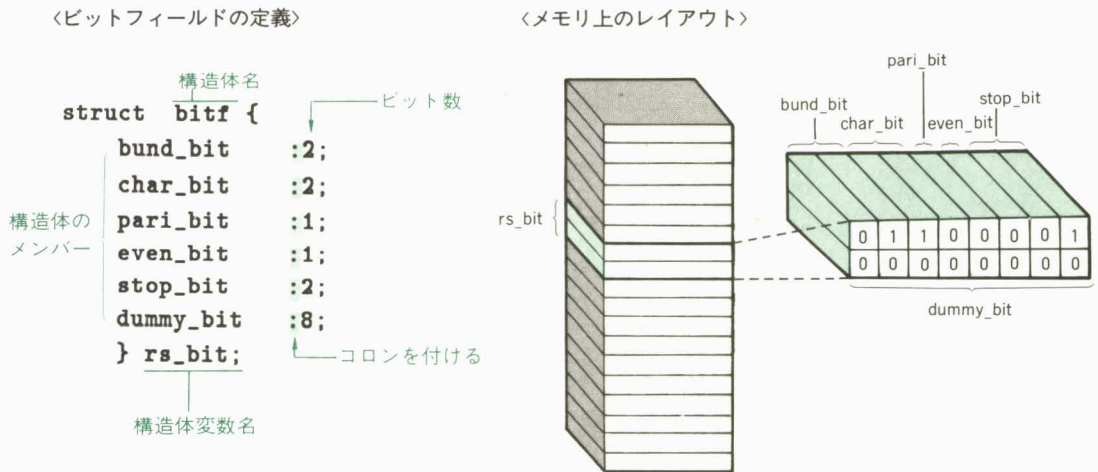
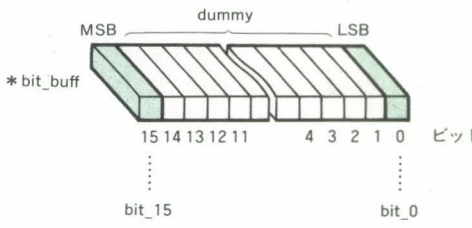



図8-16　ビットフィールドの定義

ここで，各ビットパターンを読み出したり，書き込んだりするためには，通常の構造体と同じように，

**構造体変数名.メンバー名**　　　… 通常の変数の場合
**構造体のポインター>メンバー名**… ポインタ変数の場合

という書式で参照することができます．たとえば，「char_bit」のパターンを2進数で「10b」(ビット列)としたければ，以下のように記述します．

```
rs_bit.char_bit = 2;
```

ビット列を読み込む場合でもまったく同じやり方で行います．ただし，ビットですから，かならず2進数で扱われるという点に注意してください．

このようなビットフィールドを用意しておくと，第2章のビット演算子のところで取り上げたビットのシフトやマスクを使わずに特定のビットのON／OFFを調べることが可能です．以下の**リスト8-7**に入力された数値の最上位ビット(MSB)と最下位ビット(LSB)の状態を調べるプログラムを示します．

```
 1: #include    <stdlib.h>
 2:
 3:                         符号なしであることを強調するためにunsigned で宣言する
 4: struct  bitf {
 5:     unsigned bit_Ø   : 1;
 6:     unsigned dummy   :14;
 7:     unsigned bit_15  : 1;
 8:     } *bit_buff;
 9:
1Ø:
11: main()
12:     {
13:     unsigned short    x;
14:     char      lbuff[2Ø];
15:
16:     for(;;)
17:         {
18:         printf("Enter Number : ");
19:         gets(lbuff);
2Ø:         if(lbuff[Ø] == Ø)    break; ………リターンで入力終了
21:         x = (unsigned short)atoi(lbuff);
22:
23:         bit_buff = (struct bitf *)&x; ………変数x のアドレスを構造体変数bit_buff にセット
24:
25:         printf(" HEX -----> %x¥n", x);
26:         printf(" MBS -----> %d¥n", bit_buff->bit_15);
27:         printf(" LBS -----> %d¥n", bit_buff->bit_Ø);
28:         printf("¥n");                MSB と LSB のビットの内容を表示
29:         }
3Ø:     }
```



[実行結果]

```
A>test↵
Enter Number : 1↵
 HEX -----> 1
 MBS -----> Ø
 LBS -----> 1

Enter Number : -1↵
 HEX -----> ffff
 MBS -----> 1
 LBS -----> 1
```

```
Enter Number : 0↵
 HEX -----> 0
 MBS -----> 0
 LBS -----> 0

Enter Number : 32767↵
 HEX -----> 7fff
 MBS -----> 0
 LBS -----> 1

Enter Number : 32768↵
 HEX -----> 8000
 MBS -----> 1
 LBS -----> 0

Enter Number : 65535↵
 HEX -----> ffff
 MBS -----> 1
 LBS -----> 1

Enter Number :↵    ………リターンのみで入力終了

A>
```

リスト8-7　ビットフィールドのサンプルプログラム

ビットフィールドは使いようによってはなかなか便利なのですが，次のような制限があります．

①ビットフィールドは，int型の変数のみに定義できる．
②ビットフィールドのメモリへの割り付けは，ハードウェアによって上位から行われるものと下位から行われるものがある．
③構造体で扱うので，実際の処理には時間がかかる．

とくに「int型の変数のみにしかビットフィールドを定義できない」という制限はよく忘れがちです．また，int型の変数というのはCPUによって異なりますから，C言語の特徴である「移植性の高いプログラムが書ける」ということとは矛盾してしまいます．つまり，「移植性の高いプログラムが書きたければ，ビットフィールドは使うな」というのが常識になっています．先の例のように特定のビットの状態を調べたい場合は，実際には第2章で示したビット演算子を使う方法がより多く用いられます．

# 8.5 共用体

構造体と並んで，C言語の特徴を表すものに共用体(union)があります．この共用体は，構造体と同じように複数のデータ型をひとまとめにして扱えるように考えられたデータ型であり，その書式なども構造体と同じものです．そこでここでは，共用体の概略をつかんでもらうために，まずその例題から示していくことにしましょう．

## ■共用体の考え方

共用体は，以下の**図8-17**のように定義できます．

```
┌─共用体(union)を明示する宣言子
│   ┌──────────────共用体タグ(テンプレート)
union ux{
    int   a;  ┐
    int   b;  │共用体のメンバー
    char  c;  │
    long  d;  ┘
  } uy;
  └────────共用体変数名
```

**図8-17　共用体の定義の例**

この宣言でわかるように，共用体は構造体とその書式がまったく同じかたちをしています．さてメモリ上ではどのように領域が確保されるのでしょうか．次ページの**図8-18**にそれを示します．

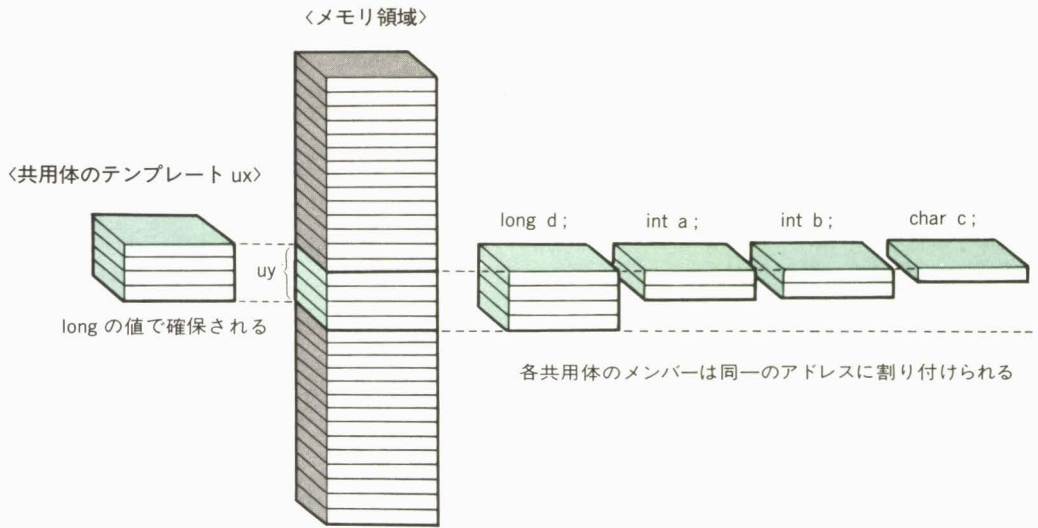
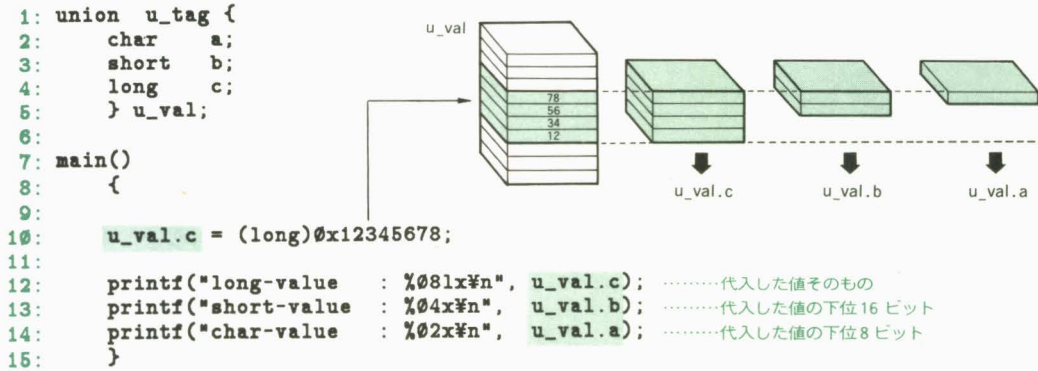
次ページの図でわかるように，共用体のテンプレートとしては，メンバーのなかの最も大きなデータサイズと同じ大きさが確保されます．そして各メンバーは，すべて共用体変数の先頭アドレスから同じように割り付けられます．この例では，メンバーのなかで最も大きな領域を占有するlong型の変数dの領域が確保され，それに重ねてint型の変数a，b，そしてchar型の変数cの領域が取られます．

共用体で定義したメンバーの参照方法は，これもまた構造体の場合と変わりません．そこで共用体のメンバーへの代入と参照を行う以下のプログラムを実行してみましょう(**リスト8-8**)．

この例でわかるように共用体のメンバーの参照には，メンバー演算子「.」を用います．また，共用体変数がポインタ変数として定義されている場合は，「－>」を用います．

図8-18　共用体で確保されるメモリ領域

```
 1: union  u_tag {
 2:      char     a;
 3:      short    b;
 4:      long     c;
 5:      } u_val;
 6:
 7: main()
 8:      {
 9:
10:      u_val.c = (long)Ø x12345678;
11:
12:      printf("long-value    : %Ø8lx¥n", u_val.c);  ………代入した値そのもの
13:      printf("short-value   : %Ø4x¥n",  u_val.b);  ………代入した値の下位 16 ビット
14:      printf("char-value    : %Ø2x¥n",  u_val.a);  ………代入した値の下位 8 ビット
15:      }
```

u_val
78
56
34
12
u_val.c
u_val.b
u_val.a

［実行結果］

```
A>test
long-value   : 12345678
short-value  : 5678
char-value   : 78

A>
```

リスト8-8　共用体の代入と参照

## ■共用体の活用

前節では共用体の概略について説明しましたが，いったいこの共用体は何の役に立つのでしょうか？ なんだかさっぱりわからないという人も多いでしょう．

実は，共用体はこれまで扱った構造体と組み合わせることで，いろいろとおもしろいことができるのです．たとえば，16ビットで表される数値の上位下位のおのおの8ビットずつをchar型の数値としてもらいたい場合などで共用体を利用することができます．これは第2章でやったようにビットのシフトやビットANDを使うことでも可能ですが，共用体や構造体を使うと以下の**リスト8-9**のようなプログラムを組むことができます．

```
 1: /*
 2:     Test procedure for UNION for short/char conversion
 3: */
 4:
 5: struct  c_tag   {      ………バイト単位で構造体を作る
 6:         char    low;
 7:         char    high;
 8:         };
 9:
10:
11: union   u_tag   {      ………上記の構造体と short の変数を共用体で重ね合せる
12:         struct  c_tag   byte_acc;
13:         short   word_acc;
14:         }
15:         u_acc;
16:
17:
18: main()
19: {
20: u_acc.word_acc = (short)0x1234;   ………共用体のメンバーに値を代入する
21:
22: printf("word value is  : %04x¥n",   u_acc.word_acc);
23: printf("high byte  is  : %02x¥n",   u_acc.byte_acc.high);
24: printf("low  byte  is  : %02x¥n¥n", u_acc.byte_acc.low);
25:                                     └─ 共用体の構造体のメンバーを参照する
26: u_acc.byte_acc.high = (char)0xFF;      (参照方法は，構造体の構造体の場合と同じ)
27:                       └高位の1バイトを0xFF に変える
28: printf("word value is  : %04x¥n", u_acc.word_acc);
29: }                                  └ 全体の値を short でのぞく
```

［実行結果］

```
A>test↵
word value is  : 1234   ………代入した short の値
high byte  is  : 12     ………short の値の上位バイト
low  byte  is  : 34     ………short の値の下位バイト

word value is  : ff34   ………上位バイトを変更した short の値

A>
```

リスト8-9 共用体のサンプルプログラム

このような構造体と共用体の組み合わせでは，構造体の構造体と同じようにメンバー演算子をつなげていくことで，各メンバーを参照します(23〜24行目)．このプログラムでは，次のようにメモリが確保されます(**図8-19**)．

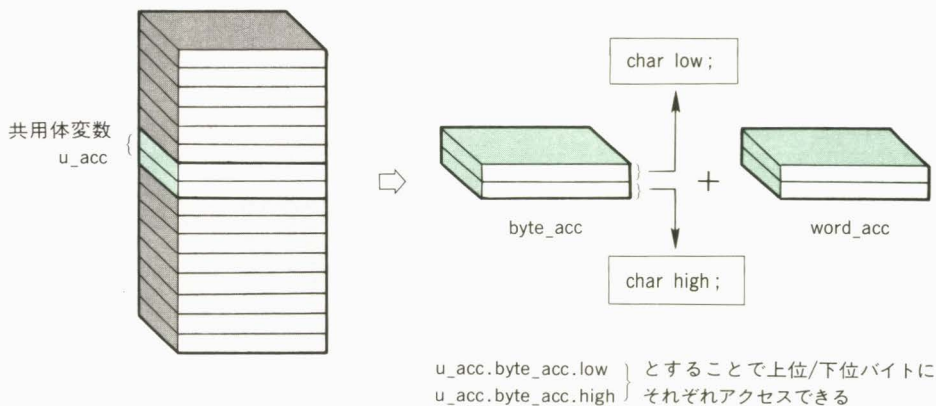



**図8-19　メモリの確保**

共用体の各メンバーは，かならず共用体変数の先頭アドレスから割り付けられますから，通常は途中のアドレスからアクセスすることはできません．しかし構造体を組み合わせると，構造体のメンバーを使って途中のアドレスの値を参照することが可能になります．これにより，**リスト8-9**で示したようなバイト単位でのデータの取り出しやマスクなどが行えます．

この例ではshortの場合を示しましたが，もちろんlongやfloat，doubleのなどのデータ型でも同じです．浮動小数点数は整数で見た場合，そのままでは当然意味をもちませんが，浮動小数点値の表現形式の変換などにはこの方法は有用です．たとえば，IEEE表記の浮動小数点形式と通常の表現形式とのコンバージョンには実際に使われている例があります．

## ■処理系に用意されている共用体

共用体は，FILE構造体と同じようにあらかじめ処理系で用意されているものがあります．たとえば，8086系の多くのC言語コンパイラに付いてくるCPUのレジスタの構造体／共用体は，構造体と共用体の組合せをうまく使った例といえるでしょう．そこで以下では，その事例を見てみましょう．

ここでは，Microsoft C Compilerの「dos.h」ヘッダファイルに含まれる構造体と共用体の定義を示します．

```
 1: /* word registers */
 2:
 3: struct WORDREGS {    ………ワード(2バイト)単位のレジスタの構造体
 4:     unsigned int ax;
 5:     unsigned int bx;
 6:     unsigned int cx;
 7:     unsigned int dx;
 8:     unsigned int si;
 9:     unsigned int di;
10:     unsigned int cflag;
11:     };
12:
13: /* byte registers */
14:
15: struct BYTEREGS {    ………バイト単位のレジスタの構造体
16:     unsigned char al, ah;
17:     unsigned char bl, bh;
18:     unsigned char cl, ch;
19:     unsigned char dl, dh;
20:     };
21:
22: /* general purpose registers union - overlays the corresponding word and
23:  * byte registers.
24:  */
25:
26: union REGS {              ………上記の2つの構造体を使った共用体
27:     struct WORDREGS x;      これで同じエリアをバイトでもワードでもアクセスすることができる
28:     struct BYTEREGS h;
29:     };
30:
31: /* segment registers */
32:
33: struct SREGS {          ………セグメントレジスタの構造体
34:     unsigned int es;      これはワード単位のみ
35:     unsigned int cs;
36:     unsigned int ss;
37:     unsigned int ds;
38:     };
```

〈注意〉
- もちろん，これがそのままレジスタに割り当てられるわけではなく，ライブラリ中でこの構造体の中身がレジスタにコピーされて使用される

図8-20　CPUのレジスタの構造体／共用体

ここで，26行目の共用体でバイト単位の構造体とワード単位（2バイト）の構造体を同じエリアに割り付けています．これによって，たとえば16ビット長のAXレジスタは，8ビット長のAH，ALという2つのレジスタとしてもアクセスすることが可能になります．

## ■共用体のまとめ

最後に共用体のまとめとして，共用体の書式と参照方法を整理しておきます．

**＜共用体の定義＞**

```
union    共用体タグ{
  メンバー1の宣言；
  メンバー2の宣言；
      ⋮
  メンバーnの宣言；
    }；
```

**＜共用体の使用宣言＞**

```
union    共用体タグ　変数名[，変数名…]；
```

**＜共用体の定義と宣言＞**

```
union    共用体タグ{
  メンバー1の宣言；
  メンバー2の宣言；
      ⋮
  メンバーnの宣言；
  }変数名[，変数名…]；
```

**＜共用体の参照＞**

```
共用体変数名.メンバー名         … 通常の変数の場合
共用体のポインター＞メンバー名… ポインタ変数の場合
```

# 第9章
# プリプロセッサと分割コンパイル

プログラムを組む場合，まず問題となるのは基本設計です．そして基本設計の段階を過ぎると，次はいかにその言語の特徴を生かしたプログラムを組んでいくかということになります．これまでの章では，そのためのC言語の言語仕様について解説を行ってきました．

C言語には，このような言語仕様だけでなくプログラミングの効率を上げるための便利な機能が備わっています．それが本章で紹介するプリプロセッサと分割コンパイルです．この2つの技法は，とくに大規模なプログラムを組む際に，そのプログラム全体の善し悪しやプログラムを組む過程での効率に大きく関わってきますから，十分に理解して効果的な使い方をしてください．

# 9.1 プリプロセッサ

C言語には，プログラムのなかでコンパイルについての指示を行う「プリプロセッサ」という機能が備わっています．この機能は，C言語そのものの仕様ではなく，コンパイラについている機能の1つと考えた方がよいでしょう．プリプロセッサへの指示は，**図9-1**に示すようにプログラムをコンパイルする前に実行されるので，その意味のとおり「前処理」と呼ばれます．

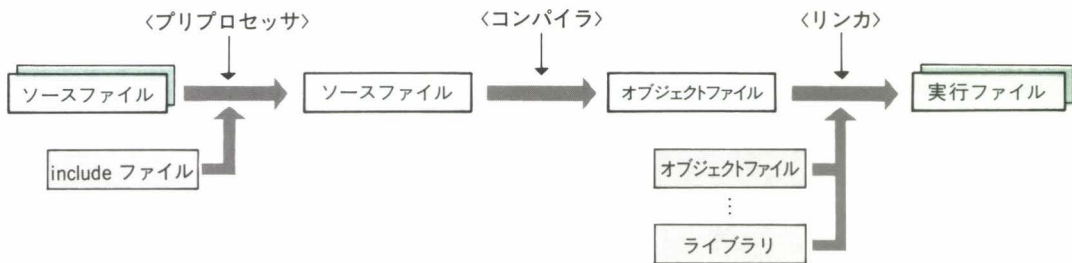



**図 9-1　コンパイルの流れ**

プリプロセッサ文の書式は，次のようになります．

**#プリプロセッサ名　……　□――最後に「；」(セミコロン)は付けない**
**└――1カラム目に書かなければならない処理系が多い**

このプリプロセッサ文は，プログラム中のどこにでも記述することができます．また，プリプロセッサ名以降は，その処理の内容に応じてファイル名や文字列などを指定します．

C言語で使うことができるプリプロセッサ文は，以下の**表9-1**に示す種類があります．

以降では，**表9-1**に示したプリプロセッサ文の詳細について表の形式でまとめていきます．

| 機能 | プリプロセッサ文 |
|---|---|
| ファイルの取り込み | # include |
| 文字列の置換/マクロ定義 | # define/ # undef |
| 条件コンパイル | # if～ # elif～ # else～ # endif |
| | # ifdef ( # ifndef) ～ # else～ # endif |
| 行番号の制御 | # line |

〈注意〉
- C言語の処理系によっては，サポートしていないプリプロセッサ文がある

**表 9-1　プリプロセッサの種類**

## ■ #include(ファイルの取り込み)

**書式**

```
#include　〈ファイル名〉
#include　"ファイル名"
```

**機能**

指定されたファイル(インクルードファイル)を#include 文の位置に取り込む

**使用例**

```
#include     <stdio.h>  ………あらかじめ指定されたディレクトリからファイルを取り込む
#include     "change.h" ………現在のディレクトリからファイルを取り込む
#include     "b:¥usr¥include¥io.h" ………パス名を明示することもできる
#include     <until.c>  ………他のソースファイルを取り込むこともできる


main(argc, argv)
    int      argc;
    char    *argv[];
    {
     :
     :
```

**コメント**

階層ディレクトリ構造を持つOSでは，取り込まれるインクルードファイルの検索方法を以下のように指定できる.

<ファイル名>　……　あらかじめコンパイラに指示してあるディレクトリ†からファイルを検索する

〝ファイル名〟　……　カレントディレクトリからファイルを検索し，見つからなかった場合あらかじめ指示してあるディレクトリを検索する

ともにファイル名として，パス名が指定された場合は，指定されたパス名のディレクトリのみ検索する

また，インクルードファイル中でも#include 文が使用できる(ネストが可能).

C言語の処理系には，「.h」の拡張子が付いたインクルードファイル(これをとくにヘッダファイルという)があらかじめ用意されており，プログラム中で関数を使用する前には必要なヘッダファイルを取り込んでおく.

† インクルードファイルを検索するディレクトリの指定方法はコンパイラによって異なる．くわしくは「入門C言語」のAPPENDIXを参照のこと.

## ■ #define(文字列の置換／マクロ定義)

**書式**

```
#define   識別子  文字列
#define   識別子(引数のリスト)  式
#undef    識別子
```

**機能**

#define 文は，その文以降に現れた識別子を文字列，または式に置換する.
#undef 文は，#define 文で指定した置換をその文以降行わない

**使用例**

```
#define    TRUE     (-1) ……………………以降の TRUE という文字列をすべて(−1)に置き換える
#define    FALSE    (!TRUE) …………先に#define で定義された文字列を使った置き換えも可能
#define    BOOL     int  ……………………typedef の代わりに#define 文を使うことも可能
#define    MULT2(X)      ((x)*(x)) ………引数を伴ったマクロ定義
                ↑
                └──空白をあけてはいけない


BOOL    func(a) ………int func (a) に置換
     int     a;
     {     ┌──────((a)*(a))に置換
     if(MULT2(a) > 10)      return(TRUE);
     else                   return(FALSE);
     }
     ⋮
     printf("TRUE is %d¥n", a); ………文字列中の「TRUE」は置換されない
     ⋮
#undef     BOOL          }
#define    BOOL    char  } BOOL を以降では char と定義し直す
#undef     MULT2   ………MULT2 を以降では置換しない
     ⋮
     ⋮
```

**コメント**

#define および #undef で指定する識別子は，ソースファイル中で置換されることを明確に示すために，通常大文字で指定する.

#define には，単純な文字列の置換と引数を伴った**マクロ置換**の2つがある．マクロ置換では，識別子を式で置き換えるだけでなく，マクロで指定した仮引数をプログラム中の実引数に変換する．ここで演算子の優先順位の関係から値は，「( )」でくくっておくことに注意する.

なお，#define による2重定義は許されていないので，同一の文字列を別の文字列に置換したい場合は，#undef で定義を取り消したあと，もう一度 #define で定義を行う.

## ■ #if～#elif～#else～#endif(条件コンパイル)

**書式**

```
#if    条件式
   ⋮
[#elif    条件式]
   ⋮
[#elif    条件式]
   ⋮
[#else]
   ⋮
#endif
```

**機能**　条件式の評価に基づき，指定したプログラム中の一部をコンパイラに引き渡す

**使用例**

```
#if   DEBUG == 1
      STACK = 100;
      switch(x)          DEBUG が 1 のとき，コンパイルされる範囲
          {
          ⋮
#elif   DEBUG == 2
      STACK = 200;
      switch(x)          DEBUG が 2 のとき，コンパイルされる範囲
          {
          ⋮
#else
      STACK = 300;
      switch(x)          DEBUG が上記以外のとき，コンパイルされる範囲
          {
          ⋮
#endif ────────── #if 文の終わりを示す
          ⋮              DEBUG の値にかかわらずコンパイルされる範囲
          ⋮
```

**コメント**

制御文の「if～else」文と同様に，条件式が真の場合，次の「#elif」,「#else」,「#endif」のいずれかが現れるまでがコンパイラに引き渡される.「#else」以降は,「#if」または「#elif」の条件が満たされない場合にコンパイルされる. ここで,「#elif」,「#else」は必要なければ省略することも可能.

また，上記の使用例で使われる「DEBUG」という定義は，プログラムの先頭で「#define　DEBUG　1」などとするか，コンパイル時にコンパイラのオプション†として指定するのが通常の方法である.

† コンパイラのオプションについては，APPENDIX を参照のこと

## ■ #ifdef( #ifndef)～#else～#endif(条件コンパイル)

**書式**

```
 #ifdef (#ifndef)     識別子
     ⋮
[#else]
     ⋮
 #endif
```

**機能**

識別子が定義されているかどうかを判断し，指定されたプログラム中の一部をコンパイラに引き渡す

**使用例**

```
#define    TAB      0x09
#define    SPACE    0x20
   ⋮
#ifdef    TAB
    printf("%c", TAB);        } 文字列「TAB」が定義されているとき，コンパイルされる範囲
        ⋮
#else
    printf("%c", 0x09);       } 上記以外のとき，コンパイルされる範囲
        ⋮
#endif ─────────────────────── # ifdef 文の終わり
   ⋮                          } 「TAB」の定義にかかわらずコンパイルされる範囲
   ⋮
#ifndef    SPACE
    printf("%c", 0x20);       } 文字列「SPACE」が定義されていないとき，コンパイルされる範囲
        ⋮
#else
    printf("%c", SPACE);      } 上記以外のとき，コンパイルされる範囲
        ⋮
#endif ─────────────────────── # ifndef 文の終わり
        ⋮                     } 「SPACE」の定義にかかわらずコンパイルされる範囲
        ⋮
```

**コメント**

「#ifdef」文は，条件が真の場合，次に現れる「#else」または「#endif」までをコンパイラに引き渡し，偽の場合「#else」から「#endif」までをコンパイラに引き渡す．ここで「#ifdef」では，#define ですでに識別子が定義されている場合，真(0以外)となり，定義されていない場合，偽(0)となる．また，「#ifndef」では条件の判断が逆になる．

なお，「#else」文は省略することが可能．

## ■ #line（行番号の制御）

| 書式 | # line　行番号　[ファイル名] |
|---|---|
| 機能 | コンパイラが出すエラーメッセージを，指定したファイル名と指定した行番号で出力する |

**使用例**

```
    :
#line    0    FUNC1       ………「# line」文の定義

int    func1(a)   ―――――この関数でエラーが起こった場合，この部分からファイル名
    char    *a               FUNC1 と 0 番からの行番号が振られる
    {
        :
    }
    :
#line    100    FUNC2     ………「# line」文の定義

void    func2(b)  ―――――この関数でエラーが起こった場合，この部分からファイル名
    char    *b;              FUNC2 と 100 番からの行番号が振られる
    {
        :
    }
    :
```

**コメント**

上記の使用例のコンパイル結果は，たとえば次のようになる．

FUNC1, line 1 : syntax error
⋮
FUNC2, line 105 : illegal character
⋮

なお，ファイル名を省略した場合は，コンパイル時に入力したファイル名が出力される．

## ■ ヘッダファイル

#include 文で取り込まれるファイルの多くは，「.h」という拡張子の付いたヘッダファイルです（「.h」でなくてもよいが，慣例としてそうなっている）．C 言語の処理系には，このヘッダファイルがあらかじめ何種類か用意されており，必要に応じてプログラムの先頭でそれらを取り込んで使用します．ヘッダファイルの中身は次のように分類できます．

・定数の定義
・関数のマクロ定義
・関数の宣言
・構造体の宣言

C言語では標準関数の多くはライブラリとして用意されていますから，関数を使用する前には，関数の宣言や定義を済ませておかなければなりません．その定義がなされているのがヘッダファイルで，プログラマは #include 文でこのファイルを取り込むだけで済んでしまいます．ただし関数によって，たとえば入出力関係ならば「stdio.h」，文字列操作ならば「string.h」というように取り込まなければならないヘッダファイルは異なります．また，ヘッダファイルを取り込む必要がない関数もあります．ヘッダファイルの中身や関数とヘッダファイルの対応表は，APPENDIX にくわしくまとめているので参照してください．

なお，ヘッダファイルは自分で作ることももちろん可能ですから，毎回行う定義などはファイルにしておくとよいでしょう．

## ■ #define 使用上の注意

「#define」文はプログラム中でよく使われるプリプロセッサです．このプリプロセッサでは，「文字列の置き換え」が行われるのであって，決して値の置き換えが行われるのではないということに注意してください．たとえば，以下の**リスト 9-1** では予想もしない結果がでてしまいます．

```
 1: #define     BUFMAX     256
 2: #define     BUF        8+BUFMAX  ………前述の BUFMAX を使った定義．一見問題なさそうだが……
 3:
 4: main()
 5:      {
 6:      int     a, b;
 7:
 8:      b = 10;
 9:      a = b * BUF;  ………BUF を使った演算
10:      printf("a -----> %d¥n", a);
11:      }

[実行結果]
A>test
a -----> 336  ……… a = b*8+BUFMAX = 10*8+256 として計算されてしまった

A>
```

リスト 9-1　#define の副作用（1）

これを防ぐためには，2行目のBUFの定義を，

```
#define   BUF   (8+NUFMAX)
```

というようにかっこでくくっておかなければなりません．また，引数を伴ったマクロ定義でも演算子の優先順位との関係で次のようなことが起こります(**リスト9-2**)．

```
 1: #define     MULT2(x)     (x * x) ………この定義はかっこで囲んでいるので問題がない?!
 2:
 3: main()
 4:      {
 5:      int     a, b, c;
 6:
 7:      b = 6;     c = 4;
 8:      a = MULT2(b + c); ………MULT2 のマクロ定義を使った演算
 9:      printf("a -----> %d¥n", a);
10:      }

[実行結果]
A>test ☑
a -----> 34 ……… a =(b+c*b+c) =(6+4*6+4) として計算されてしまった

A>
```

リスト9-2　#define の副作用(2)

この場合も，以下のように仮引数をかっこで囲んでおきます．

```
#define   MULT2(x)     ((x)*(x))
```

このように#defineによる文字列の置換は，ときとして予想もしない結果をもたらすことがあるので十分注意する必要があります．ここでの最善の手段は，「#defineで数値を定義する場合，数値として扱う文字列をかっこでくくっておく」ということです．

## ■マクロ定義と関数

C言語の関数のなかには，純粋にライブラリ関数として定義されているものと，ヘッダファイルのなかで「#define」文によりマクロ定義されているものの2種類があります．たとえば以下の2つは，「ファイルから1文字読み込む」という同一の機能を持った関数です．

fgetc(fp) …… ライブラリ関数
getc(fp) …… 「stdio.h」のなかで定義されているマクロ

さて，この2つはどのように使い分ければよいのでしょうか？ 以下の図9-2を見てください．

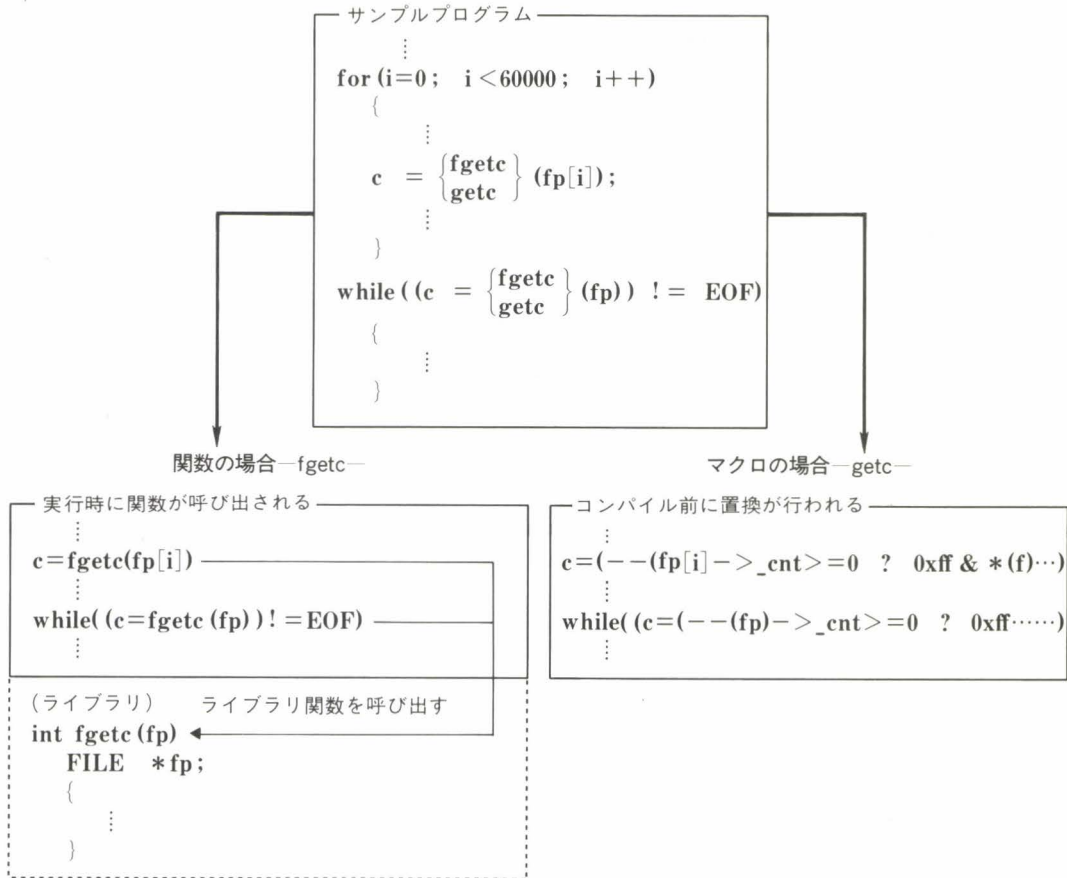



図9-2 関数とマクロ定義の違い

関数の場合は，その実体がリンクされるライブラリのなかにあり，関数呼び出しという形式で実行されます．つまり，多くの箇所で関数呼び出しが行われても，その実体は1つなのでオブジェクトファイルは大きくなりません．しかし，呼び出す際の引数のやりとりなどでオーバーヘッドが高くなり，実行速度が落ちる可能性があります．

マクロ定義の場合は，コンパイルされる前に関数が展開されます．つまり，多くの箇所でその関数が使用されると，すべて同じように置換が起こりますからファイルのサイズは大きくなります．しかし，逆に関数呼び出しのようなオーバーヘッドはありません．

このように，関数とマクロでは実行方法に違いがでてきます．通常のプログラムでは，それほど気にする必要はありませんが，実行速度やプログラムサイズを問題とするプログラムを書く際には，その違いに注意してください．

また，このことは自分でプログラムを作る場合にもまったく同様です．ただし，マクロ定義の場合は，そのなかで変数の宣言などを行うといったことはもちろんできませんから，短く単純な式に限られることはいうまでもありません．

## ■#if とデバッグ

どんなプログラムでも，デバッグを必要としないプログラムはありません．このデバッグを効率よく行い，かつできあがった実行ファイルからは，その痕跡が残らないようにする方法の１つとして，「#if」や「#ifdef」による条件コンパイルがあります．

まず，プログラムの一番最初に，

```
① #define  DEBUG     1
```

という１行を入れておきます．この１行を付ける代わりに，コンパイラの起動に際して，たとえば次のようにオプションを指定すると，この１行が含まれたように扱ってくれるコンパイラもあります(コンパイラのオプションについては APPENDIX を参照のこと)．

```
② A>cc   -DDEBUG    test.c
          └─プリプロセッサへ文字列の置換を指示するオプション
```

通常は「－D識別子＝文字列」という形式で指定するが，識別子のみを指定することもできる．

こうしておいて，プログラムのネックとなりそうな箇所にいくつか，

```
{ #if          DEBUG  ……  ①の方法 }
{ #ifdef       DEBUG  ……  ②の方法 }
      printf("X= %d, Y= %d, Z= %d¥n" , x, y, z);
  #endif /* DEBUG */
```

というような数行を入れておくわけです．デバッグというのは，つきつめてみれば，「どの時点で，どんな値が，どこに入っているか」を確かめることですから，こういった方法が最も確実です．

こうしてデバッグが終了したら，プログラムの先頭の #defineの定義で「１」の値を「０」とするだけで(または，通常の方法でコンパイルするだけで)，その部分はコンパイルされなくなるので，実行ファイルにはデバッグの情報は現れません．

小規模なプログラムであれば，これらの「#if～」も，あまり格好のよいものではありませんから消

しておくことも多いのですが，大規模なプログラムでは「#if～」を消すのもひと苦労なので，普通はそのまま残しておきます．よく他人の作ったソースファイルを見ると，この「#if～」がデバッグで使われたまま残っているのを見かけます．こういうものを見かけた場合は他人のプログラムの組み方を学ぶチャンスですから，この #if のあたりを中心にリストを眺めてみましょう．きっと，なんらか得られるものがあるはずです．また，#if による条件コンパイルは，デバッグだけでなくコンパイラの差異を吸収するためにも使われます．その例を次のサンプルプログラムで示すことにしましょう．

## ■サンプルプログラム

プリプロセッサを使ったプログラムを以下に紹介します．これは，UNIX 上と MS-DOS 上で共通に動くように書かれたプログラムです．

プログラムの最初の行にあるシステム名のマクロ定義の「0」と「1」を書き替えることによって，どちらのシステムにも対応させています.実際はここまで派手にプリプロセッサを使うことはない(もし，使わなければならない場合は，もう1つ別のプログラムを作る)はずですが，いちおう「こんなふうに使うんだ」というところを理解してください．

このプログラムはキーボードから打った1文字を16進数表示に変えて，画面に出力するものです．MS-DOS 上の Microsoft C Compile や Lattice C の場合は，このようにコンソールから直接入出力するのための関数が付いていますが，UNIX にはありません．UNIX では，キーボードもディスクも「デバイス」として同一に扱われているので，常に入出力動作はバッファリングされてしまいます．つまり，1行の入力によって，初めてそれまでの数文字が入力されるのです．また，これは画面への出力についてもまったく同じことがいえます．

そこで1文字キーボードから打った場合，すぐにその1文字がプログラムに入力されるように，またプログラムから出力した1文字がすぐに画面に表示されるように，MS-DOS とは違った細工をしてやらなければなりません.その用意をしているのが14行目から33行目までの部分です(くわしい説明は，UNIX のマニュアルを参照してください)．

〈注意〉UNIX は標準の C コンパイラ(SYSTEM III 以上)，MS-DOS 上では MS-C を使ってください

```
 1: /*
 2: Display Real-Time Key-Shot in Hexadecimal
 3:         UNIX -- SYSTEM III/V & 4.X BSD        MS-DOS -- Microsoft C
 4: */
 5: #define     UNIX        Ø
 6: #define     MSDOS       1
 7: #include    <stdio.h>
 8: #define     ESC     Øx1B
 9:
1Ø: #if     MSDOS
11: #include    <conio.h>
12: #endif
13:
```

- 5～6行目：UNIX の場合「1」／そうでない場合「0」、MS-DOS の場「1」／そうでない場合「0」 ― UNIX か MS-DOS かを決めるスイッチ
- 7～8行目：UNIX でも MS-DOS でも共通なヘッダファイルとマクロ定義
- 11行目：MS-DOS に必要なヘッダファイル ― コンソールから1文字入力を行う getch 関数が定義されている

```
14: #if      UNIX
15: #include     <sgtty.h>  ……………ターミナルと入出力
16:                                   を行うための構造体
17: struct  sgttyb  oldf;             が定義されている
18: struct  sgttyb  newf;
19:
20: setr()
21: {
22:      ioctl(0, TIOCGETP,  &oldf);
23:      ioctl(0, TIOCGETP,  &newf);          UNIX に必要な
24:      newf.sg_flags |= (RAW);              ヘッダファイル
25:      newf.sg_flags &= ~ECHO;              と関数の定義
26:      ioctl(0, TIOCSETP, &newf);
27: }
28:
29: resetr()
30: {
31:      ioctl(0, TIOCSETP, &oldf);
32: }
33: #endif
34:
35: main()
36:      {
37:      int c;
38:      puts("Strike the key to display the Key-Codes in HexaDecimal.");
39:      puts(" >>>>>>> End of program : Perss ESC-Key.");
40: #if      UNIX
41:      setr();     UNIX の場合，実行される関数
42: #endif
43:      while(c != ESC)
44:           {
45: #if      UNIX
46:           c = getchar();     UNIX の場合，実行される関数
47: #endif
48:
49: #if      MSDOS
50:           c = getch();     MS-DOS の場合，実行される関数
51: #endif
52:           printf(" 0x%2x", 0x00FF & c);
53:           }
54: #if      UNIX
55:      resetr();
56:      puts(" ");     UNIX の場合，実行される関数
57: #endif
58:      }
```

［実行結果］

```
A>test
Strike the key to display the Key-Codes in HexaDecimal.
 >>>>>>> End of program : Perss ESC-Key.;
 0x20 0x31 0x32 0x33 0x1b ………最初にスペース，次に数字の 1，2，3，ESC とキーを押した

A>
```

リスト 9-3　UNIX／MS-DOS 共用プログラム

# 9.2　分割コンパイル

少し大きなプログラムを書くようになると，かならず使われるのがここで紹介する分割コンパイルという手法です．分割コンパイルは，プログラムのモジュール化と深い関係があり，効率的なプログラム開発を進める上で欠かすことができません．

## ■ プログラムのモジュール化

モジュールとは，いってみれば部品(パーツ)の集まりのことで，その組み合わせが1つの物 — たとえばパソコン — を作ります．同じようにC言語のプログラムでも，関数の集まりが1つの実行ファイルを作ることになるわけです．

ここでよい部品(関数)とは，第1章でも解説したように見やすさ(1画面以内に収まること)と機能のわかりやすさにあります．また，その関数の内容を変更した場合でも他の関数に影響を与えないように設計することが重要です．つまり，他に依存する部分を極力少なくし，関数としての独立性を保たなければなりません．

さて，このような関数を組み合わせてプログラムを作っていくわけですが，大きなプログラムの場合，それが1つのソースファイルになっていると非常に効率が悪くなります．たとえば，ちょっとした修正をしようとすると(よくあることです)，エディタで修正箇所を捜すのも手間がかかり，たった1ヶ所の修正でも，必要のない関数まで再コンパイルする必要があります．

そこで，ソースファイルをいくつかの部分に分割しておけば，むだなコンパイル — すでにバグがないとわかっている部分をコンパイルしなおす — が減ってきて，コンパイルにかかる余分な時間を節約することができます．

このようにプログラムを関数によって分類し，機能別にまとめることを**プログラムのモジュール化**と呼びます．たとえば，

- ・入出力を受け持つ関数の集まり
- ・画面表示を受け持つ関数の集まり
- ・メインルーチン

- ・処理系に依存した関数の集まり
- ・OS に依存した関数の集まり
- ・処理系にも OS にも依存しない関数の集まり

というようなモジュール別にファイルを分けておきます．こうしておけば，コンパイル時間の節約だけでなく，デバッグや移植性を高めるうえでも非常に有益であるということがわかるでしょう．

## ■分割コンパイルの手法

C言語でモジュール化を支援するのが，ここで紹介する分割コンパイルという手法です．まず，コンパイルとリンクについて簡単におさらいしておきましょう†．まず，以下に実行ファイルができるまでを図にしてみます(図9-3)．

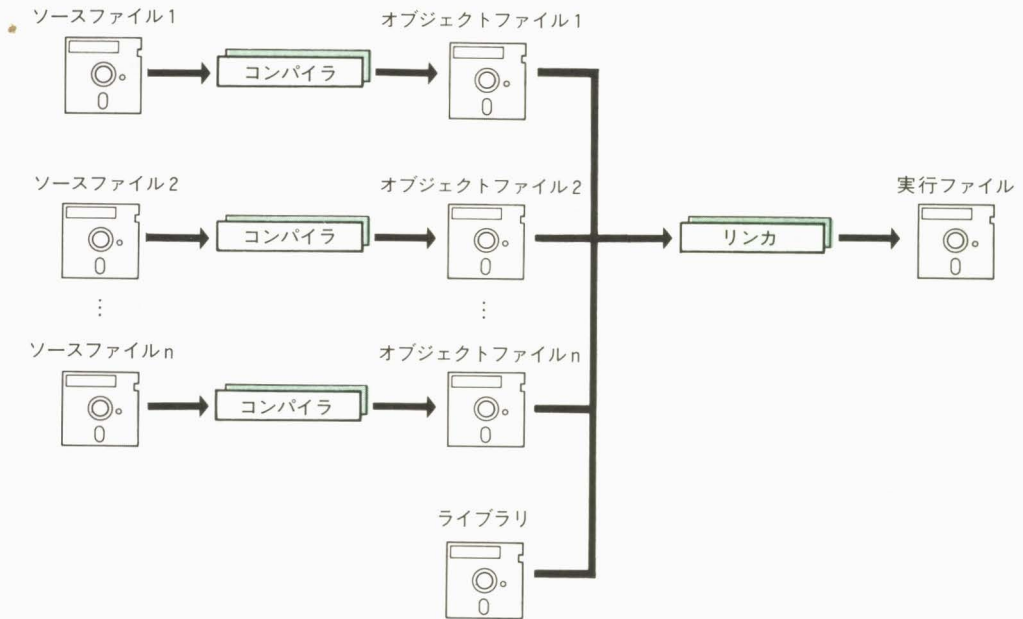



図9-3　実行ファイルができるまで

この図でわかるように，「コンパイル」と「リンク」の2つの過程があること自体，すでに分割コンパイルが可能であることを意味しています．実際に分割コンパイルを行うために必要な知識は以下の3つです．

① コンパイルの方法
② リンクの方法
③ 異なるコンパイル単位間での変数と関数の受渡しの方法

①と②については通常の方法とほとんど変わりませんが，③の項目がプログラムを組む上で最も重要なポイントになります．

† コンパイルとリンクについては，前書「入門C言語」でくわしく解説している．

## ■コンパイルとリンク

コンパイルの方法は，1つのソースファイルでも複数のソースファイルでも変わりません．分割コンパイルでは，リンクする際に複数のオブジェクトファイルを指定するだけでよいのです．以下にMicrosoft C Compilerを使った分割コンパイルの実行例を示します(図9-4)．

```
A>msc b:test0.c, b:test0.obj /Gs/G0/Gt/Od/Zd/AS/W1/Zp/J;
Microsoft (R) C Compiler  Version 3.00.17
Copyright (C) Microsoft Corp 1984, 1985.  All rights reserved.

A>dir b:

 ドライブ B: のディスクにはボリュームラベルがありません
 ディレクトリは B:¥

TEST0    C        310  86-09-01   5:39
TEST1    C        180  86-09-01   5:45
TEST0    OBJ      432  86-09-01   6:05
TEST1    OBJ      340  86-09-01   5:56
        4 個のファイルがあります
  1246208 バイトが使用可能です

A>link b:test0.obj+b:test1.obj, b:test.exe, NUL, EM+SLIBFP+SLIBC/NOD;

Microsoft 8086 Object Linker
Version 3.01 (C) Copyright Microsoft Corp 1983, 1984, 1985

A>dir b:

 ドライブ B: のディスクにはボリュームラベルがありません
 ディレクトリは B:¥

TEST0    C        310  86-09-01   5:39
TEST1    C        180  86-09-01   5:45
TEST0    OBJ      432  86-09-01   6:05
TEST1    OBJ      340  86-09-01   5:56
TEST     EXE     6066  86-09-01   6:06
        5 個のファイルがあります
  1240064 バイトが使用可能です

A>
```



図9-4 Microsoft C Compilerによる分割コンパイルの例

これらのコンパイルとリンクの方法は，他の処理系でもまったく同じことです(各処理系による違いは，「入門C言語」のAPPENDIXで取り上げている)．C言語によっては，コンパイラとリンカを制御するコマンドが付属しているものがあり，その場合はもっと簡単に分割コンパイルを行うことができます．また，各自でコンパイル用のバッチファイルを作っておくのもよいでしょう．

なお，この分割コンパイルによるプログラム開発を自動化するツールとして「make」と呼ばれるソフトウェアがあります(開発ツールについては応用編で取り上げる)．

## ■変数と関数の受渡し

分割コンパイルをする際の最も大きな問題は，異なるコンパイル単位間での変数と関数の受渡しです．これは変数や関数の有効範囲と密接な関係があります(「4.5 変数の有効範囲」でくわしく解説しているので参照のこと)．まず，コンパイル単位間で共通に使う値の宣言と他のコンパイル単位で宣言された値を利用する場合の宣言について，**表9-2**にまとめます．

| | 共通に使う値の宣言 | 利用する側の宣言 |
|---|---|---|
| 変数 | グローバル変数 | extern |
| 関数 | —— | extern （省略可能） |

**表9-2　受渡しを行うための変数と関数の宣言**

コンパイル単位で共通に使う変数は，**グローバル変数**として宣言します．関数の場合は，つねにグローバルなので宣言の必要はありません．この変数や関数を別のコンパイル単位で利用するには，記憶クラスとしてexternを宣言します．extern宣言は他のコンパイル単位にその実体があることを示し，リンク時にリンカによってその参照が解決されます．以下にこれらの関係を図にしておきます(**図9-5**)．

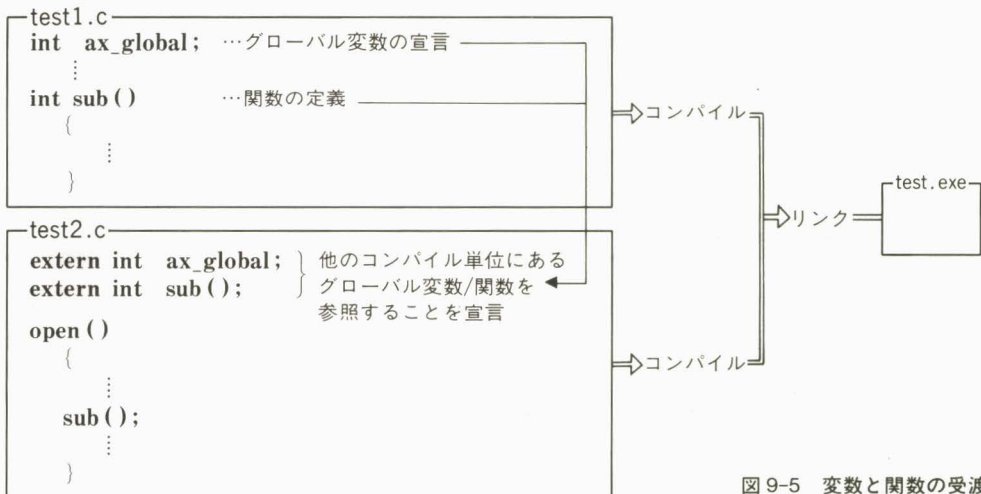



**図9-5　変数と関数の受渡し**

このように関数や変数は実体があるので，別のコンパイル単位から参照できるのですが(このことを**外部参照**といいます)，#defineのマクロ定義はその実体がなく，コンパイラを通る前に展開されてしまうので，外部参照はできません．

## ■サンプルプログラム

それでは，実際にプログラムを組んで，分割コンパイルを試してみましょう．以下の**リスト9-4**は，それぞれ1つの関数を持つ2つのコンパイル単位として構成されています．実際には，このような短いプログラムを分割コンパイルすることはありませんが，ここでは変数と関数の受渡しに注目して見てください．

ソースファイル test0.c

```
 1: /*
 2:      TEST0.C   TEST-PROGRAMS <MAIN>
 3: */
 4:
 5: #include    <stdio.h>
 6:                          ← グローバル変数
 7: int      ix;
 8: char     hexbuf[10];
 9:
10: extern  int    convhex();
11:                          → convhex 関数は，このモジュール
12:                            ではなく外部にある
13: main()
14:      {
15:      int i;
16:
17:      for(i = 0 ; i < 20 ; ++i)
18:          {
19:          ix = i;
20:
21:          convhex(); ………外部関数の実行
22:
23:          printf("Nnmber is %s¥n",hexbuf);
24:          }
25:      }
```

ソースファイル test1.c

```
 1: /*
 2:      TEST1.C HEX-CONVERT ROUTINES
 3: */
 4:
 5: #include    <stdio.h>
 6:                          → グローバル変数の参照を宣言
 7: extern  char    hexbuf[];
 8: extern  int     ix;
 9:                          convhex 関数の定義(通常のやり方でよい)
10: convhex()
11:      {
12:      sprintf(hexbuf, "%04x¥0", ix); ………グローバル変数hexbuf に同じくグローバル変数ix の
13:      }                                   変換結果を入れる
```

［コンパイルとリンク］

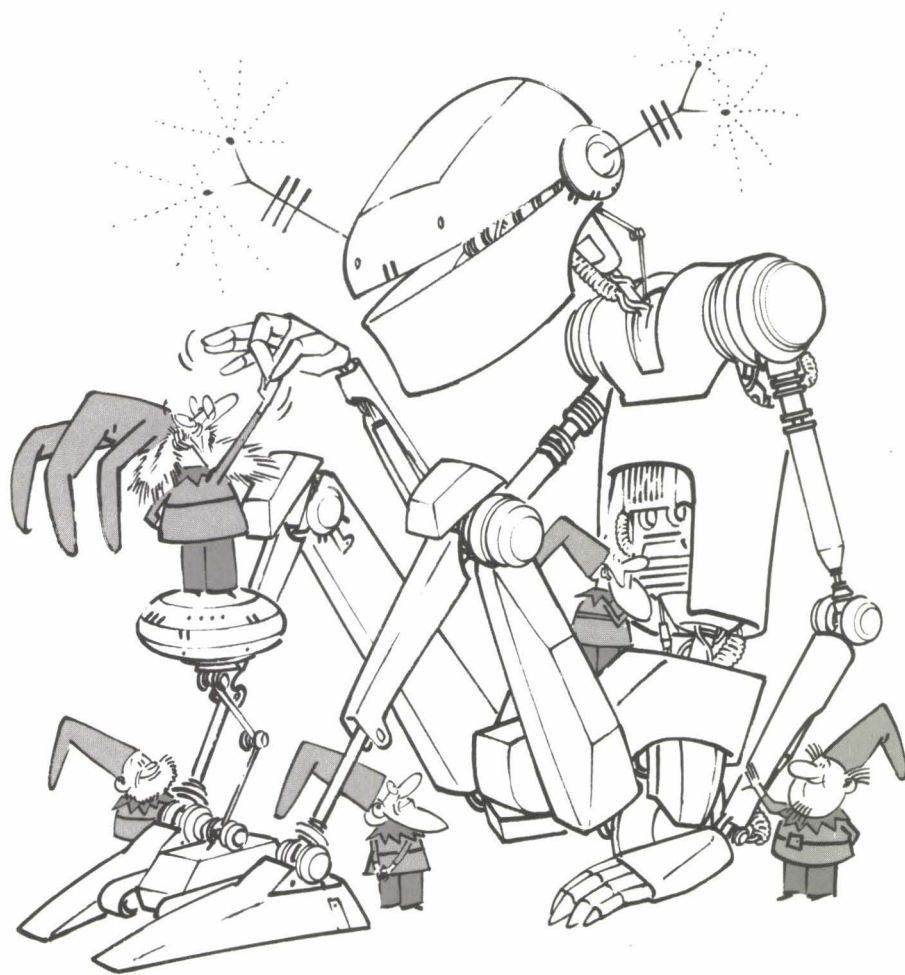
図9-4で示す方法でコンパイルとリンクを行い，「test」という実行ファイルを作成する（Microsoft C Compilerの場合）

```
A>msc b:test0.c, b:test0.obj /Gs/G0/Gt/Od/Zd/AS/W1/Zp/J;
Microsoft (R) C Compiler  Version 3.00.17
Copyright (C) Microsoft Corp 1984, 1985.  All rights reserved.


A>msc b:test1.c, b:test1.obj /Gs/G0/Gt/Od/Zd/AS/W1/Zp/J;
Microsoft (R) C Compiler  Version 3.00.17
Copyright (C) Microsoft Corp 1984, 1985.  All rights reserved.


A>link b:test0.obj+b:test1.obj, b:test.exe, NUL, EM+SLIBFP+SLIBC/NOD;

Microsoft 8086 Object Linker
Version 3.01 (C) Copyright Microsoft Corp 1983, 1984, 1985
```

［実行結果］

```
A>test
Nnmber is 0000
Nnmber is 0001
Nnmber is 0002
Nnmber is 0003
Nnmber is 0004
Nnmber is 0005
Nnmber is 0006
Nnmber is 0007
Nnmber is 0008
Nnmber is 0009
Nnmber is 000a
Nnmber is 000b
Nnmber is 000c
Nnmber is 000d
Nnmber is 000e
Nnmber is 000f
Nnmber is 0010
Nnmber is 0011
Nnmber is 0012
Nnmber is 0013

A>
```

リスト9-4　分割コンパイルのサンプルプログラム

# 第10章
# プログラム開発の事例

この章ではＣ言語による実用的なプログラムをいくつか紹介しましょう．パソコン上のＣ言語でのアプリケーション開発は，規模が大きいといっても大型コンピュータのものとは違って，どちらかというとアセンブラのような使われ方のものが多いようです．つまり「ハードウェア寄り」と言われているプログラムが多いのです．そこで，この章でもハードウェアに依存したプログラムを１つ取り上げました．

最初に紹介するのは「XREF」です．このプログラムは，関数や変数が使われている行番号のリストを作ります．大きなプログラムのデバッグには必需品であり，またとくにＣ言語ではよく使われるようです．

次は「LESS」という逆スクロールができる「more」コマンドを作ってみました．このコマンドを使うとファイルの中身を見る際に，まるでエディタを使っているように上下スクロールができるのでたいへん便利です．

# 10.1 XREF — 簡易型クロスリファレンスリスト生成プログラム

## ■ プログラムの概要

このプログラムは，C 言語や BASIC で書かれたソースファイルのクロスリファレンスリストを生成します．クロスリファレンスリストはプログラム中の関数や変数名を切り出してくれるので，プログラムの解析を行ったり，ドキュメントを書く際などに役に立ちます．

このプログラムでとくに重要な部分は，トークン(変数名や関数名)の切り出しを行っている「getto-ken 関数」と，切り出したデータを線形リストとして管理しているところでしょう．また，データの検索には関数の再帰呼び出しと木構造も用いられています．そのための手法として，C 言語の特徴である「ポインタによる変数管理」もかなり複雑な形で行っていますので，じっくり解析して各自でプログラミングする際の参考にしてください．

## ■ プログラムリストと実行結果

```
 1: /*
 2:      xref.c - cross reference (binary tree)
 3: */
 4:
 5: #ifndef lint
 6: static char sccsid[] = "@(#)xref.c  2.3 8/8/86";
 7: #endif
 8:
 9: #include     <stdio.h>   } ヘッダファイルの取り込み．LSI Cの場合，11行目以降に { # include <stdlib.h>
10: #include     <ctype.h>   }                                                   { # include <string.h>  } の3行を追加
                                                                               { # pragma nonrec
11:
12: #ifdef MSC  ………MS-Cの場合，起動時に-Dオプションで指定してもよい
13: #include     <errno.h>   } MS-C の場合，これらのヘッダファイルも取り込む
14: #include     <fcntl.h>   }
15: #endif
16:                                                       ……malloc 関数を使いやすくするためのマクロ
17: #define    xalloc(t, n)      (t*)malloc((unsigned)sizeof(t)*(n))
18: #define    xfree(p)          free((char *)(p))  ……………free 関数を使いやすくするためのマクロ
19: #define    iscsymf(c)        (isalpha(c)||(c)=='_')  ……トークンの最初の文字を識別するマクロ
20: #define    iscsym(c)         (isalnum(c)||(c)=='_')  ……トークンの2文字目以降を識別するマクロ
21:
22: #define    MAXSYMB     256   ……………シンボルの最大長を表す
23: #define    DFLSYMB     7     ……………シンボルを表示する際のデフォルトの長さ
24: #define    DFLNAME     7     ……………ファイル名を表示する際のデフォルトの長さ
25: #define    DFLNUMB     3     ……………行番号を表示する際のデフォルトの長さ
26: #define    LESS        (-1)  }
27: #define    GREAT       1     } strcmp 関数を使った際に返ってくる値の定義
28: #define    EQUAL       0     }
```

```
29:
30: struct _line { ………行番号の情報を管理する構造体
31:     struct _line *tail;
32:     long    number;
33: };
34:
35: struct _link { ………ファイル名とそのファイル内での行番号を管理する構造体
36:     struct _link *next;
37:     struct _line *head;
38:     char    *name;
39: };
40:
41: struct _node { ………最終的なシンボルをためておく構造体
42:     struct _node *left;  } ツリー構造を管理するための左(小さい方)と右(大きい方)への自己参照構造体への
43:     struct _node *right; } ポインタ
44:     struct _link *list;
45:     char    *symbol;
46: };
47:
48: typedef struct _line line; }
49: typedef struct _link link; } 各構造体の typedef を行う
50: typedef struct _node node; }
51:
52: char    *malloc();………malloc 関数の使用宣言
53: char    *progname;………このプログラムの実行時の名前
54: int      symblen; ………扱ったシンボルの最大長，表示の際の桁揃えに使う }
55: int      namelen; ………処理したファイル名の最大長                     } この3つの変数はすべて表示の際の桁そろえのためのもの
56: int      numblen; ………最大の行番号の log10 をとったもの            }
57: char    *filename;………現在処理中のファイル名
58: long     lineno;  ………現在処理中の行番号
59: int      lastchar;………最後に読み込んだ1文字
60: char     token[MAXSYMB+1];………gettoken 関数で切り出したトークンを入れておく配列
61:
62: int gettoken(fp) ………トークンの切り出しを行う関数
63: FILE    *fp;
64: {
65:     int c, i;
66:
67:     for (c = lastchar; c != EOF; c = getc(fp)) { ………トークンの先頭を見つける
68:         if (!iscsym(lastchar) && iscsymf(c))
69:             break;
70:         if (c == '¥n')
71:             lineno++; ………行番号の格納
72:         lastchar = c;
73:     }
74:     for (i = 0; i < MAXSYMB; i++) { ………トークンの切り出し
75:         if (c == EOF || !iscsym(c))
76:             break;
77:         token[i] = c; ………トークンの配列にセットする
78:         c = getc(fp);
79:     }
80:     while (c != EOF && iscsym(c)) ………トークンが長すぎる場合，終わりを読み捨てる
81:         c = getc(fp);
82:     lastchar = c;
```

```
 83:     token[i] = '¥Ø';………最後にヌル文字を入れる
 84:     if (i > symblen)
 85:         symblen = i;    } トークンの長さの最大値の記録
 86:     return (i);
 87: }
 88:
 89: void    nocore() ……… エラーが発生したらプログラムから抜ける関数
 9Ø: {
 91: #ifdef MSC
 92:     extern int  errno;      perror 関数のない処理系では,
 93:                               fprintf (stderr, "%s : Not enough core¥n", progname) ;
 94:     errno = ENOMEM;         とする
 95: #endif
 96:     perror(progname);
 97:     exit(2);
 98: }
 99:
1ØØ: char    *strsave(s) ………切り出した文字列を新しい領域にセーブしておく関数
1Ø1: char    *s;
1Ø2: {
1Ø3:     char    *p;
1Ø4:
1Ø5:     if ((p = xalloc(char, strlen(s)+1)) == NULL)
1Ø6:         nocore();
1Ø7:     strcpy(p, s);
1Ø8:     return (p);
1Ø9: }
11Ø:
111: line    *mkline(p) ………_line 構造体のデータ部分を作る関数
112: line    *p;
113: {
114:     line    *q;
115:
116:     if ((q = xalloc(line, 1)) == NULL)
117:         nocore();
118:     q->tail = p; ………テイルポインタの記録
119:     q->number = lineno;………現在処理中の行番号の記録
12Ø:     return (q);
121: }         └── 構造体へのポインタを返す
122:
123: link    *mklink(p) ………新しい _link 構造体の作成
124: link    *p;
125: {
126:     link    *q;
127:
128:     if ((q = xalloc(link, 1)) == NULL)
129:         nocore();
13Ø:     q->next = p;
131:     q->head = mkline((line *)NULL);   } 各ポインタの代入と最後を表す NULL ポインタの書き出し
132:     q->name = filename;
133:     return (q);
134: }         └── 構造体へのポインタを返す
135:
136: node    *mknode(s)………新しいノードを作る関数
```

```
137: char    *s;
138: {
139:     node    *p;
140:
141:     if ((p = xalloc(node, 1)) == NULL)
142:         nocore();
143:     p->left = p->right = NULL;              ┐
144:     p->list = mklink((link *)NULL);         │ 各ポインタの代入と最後を表わすNULLポインタの書き出し
145:     p->symbol = strsave(s);                 ┘
146:     return (p);
147: }
148:
149: int sign(n) ……… strcmp関数からの符号を判断する関数
150: int n;
151: {
152:     return (n < 0 ? LESS: n > 0 ? GREAT: EQUAL);
153: }
154:
155: node    *mktree(p) ……… データのツリー構造を作るための関数．LSI Cでは154行目に「recursive」を追加
156: node    *p;
157: {
158:     if (p == NULL) ……… データが最初または末端のとき
159:         p = mknode(token);
160:     else
161:         switch (sign(strcmp(token, p->symbol))) { ……… トークンの検索
162:         case LESS:
163:             p->left = mktree(p->left);      ┐ 小さい場合は，ツリーの左側へ移動
164:             break;                          ┘
165:         case GREAT:
166:             p->right = mktree(p->right);    ┐ 大きい場合は，ツリーの右側へ移動
167:             break;                          ┘
168:         case EQUAL:
169:             if (p->list->name != filename)  ┐ 登録済みで同一ファイルでなければ
170:                 p->list = mklink(p->list);  ┘ ファイルのリストをのばす
171:             else if (p->list->head->number != lineno)     ┐ 同一行でなければ
172:                 p->list->head = mkline(p->list->head);    ┘ 行番号のリストをのばす
173:             break; ……… 同一ファイル，同一行であれば新たな登録は行われない
174:         }
175:     return (p);
176: }
177:
178: int xlog10(n) ……… log10の値を算出する関数
179: long    n;
180: {
181:     int i;
182:
183:     for (i = 0; (n /= 10) > 0; i++)
184:         ;
185:     return (i);
186: }
187:
188: node    *xref(fp, p) ……… このプログラムの本体．ここで最初のmktree関数が呼ばれ，
189: FILE    *fp;                  ツリーが作られる
190: node    *p;
```

```
191: {
192:     int length;
193:
194:     lineno = 0L;           } 初期値の設定（各ファイルごとに初期化する）
195:     lastchar = '¥n';       }
196:     while (gettoken(fp))
197:         p = mktree(p); ………mktree 関数の呼び出し
198:     if ((length = xlog10(lineno) + 1) > numblen) } 処理終了後，行番号の最大値を取り出す
199:         numblen = length;                        }
200:     return (p);
201: }
202:
203: void    prline(p) ………クロスリファレンスリストの行番号の表示．LSI Cでは202 行目に「recursive」を追加
204: line    *p;              再帰を使ってポインタのチェインをたどり出力を行う
205: {
206:     if (p != NULL) {
207:         prline(p->tail);
208:         printf(" %*ld", numblen, p->number);
209:     }
210: }
211:
212: char    *prlink(p, s) ……クロスリファレンスリストのファイル名の出力．LSI C では211 行目に「recursive」を追加
213: link    *p;                ここでも再帰を使ってチェインをたどっている
214: char    *s;
215: {
216:     if (p != NULL) {
217:         s = prlink(p->next, s);
218:         printf("%-*s", symblen, s);
219:         if (p->name != NULL)
220:             printf(" %-*s", namelen, p->name);
221:         prline(p->head);
222:         printf("¥n");
223:         s = " ";
224:     }
225:     return (s);
226: }
227:
228: void    prnode(p) ………ツリーを再帰によって，左側から右側へ移動しながら，prlink 関数によって，シンボ
229: node    *p;              ル，ファイル名，行番号を表示させる．LSI C では227 行目に「recursive」を追加
230: {
231:     if (p != NULL) {
232:         prnode(p->left);
233:         prlink(p->list, p->symbol);
234:         prnode(p->right);
235:     }
236: }
237:
238: void    fatal(name) ………ファイルがオープンできない場合，エラーとファイル名を出力する
239: char    *name;
240: {
241:     fputs(progname, stderr); } perror 関数のない処理では，
242:     fputs(": ", stderr);     }   fprintf (stderr, "%s: %s: No such file¥n", progname, name);
243:     perror(name);            } とする
244: }
```

```
245:
246: void    main(argc, argv) ……… メイン関数．LSI C では 245 行目に「# define MSDOS」を追加
247: int argc;
248: char    **argv;
249: {
250:     FILE    *fp, *fopen();
251:     node    *root;
252:     int length;
253:
254: #ifdef MSDOS
255:     progname = "xref";
256: #else
257:     progname = argv[0]; ………プログラム名のストア
258: #endif
259:     symblen = DFLSYMB;
260:     namelen = DFLNAME;
261:     numblen = DFLNUMB;
262:     root = NULL;…………シンボルのツリーを保持しておくための変数 root の初期化
263:     switch (argc) {
264:     case 1:  …………………入力された引数が 1 つならば，入力ファイルを標準入力とする
265: #ifdef MSC ………ラージモデルでコンパイルする場合，バイナリモードを使用する(MS-C のライブラリのバグ)
266:         setmode(fileno(stdin), O_BINARY);              以下の 2 つの # ifdef も同じ
267: #endif
268:         filename = NULL; ………標準入力の場合，ファイル名は表示しない
269:         root = xref(stdin, root);
270:         break;
271:     case 2: ………引数が 2 つなら，ファイル名をオープンし，処理を行う
272: #ifdef MSC
273:         if ((fp = fopen(argv[1], "rb")) == NULL) {
274: #else
275:         if ((fp = fopen(argv[1], "r")) == NULL) {
276: #endif
277:             fatal(argv[1]);
278:             break;
279:         }
280:         filename = NULL; ………ファイル名が 1 つの場合，ファイル名は表示しない
281:         root = xref(fp, root);
282:         fclose(fp);
283:         break;
284:     default: ………引数が 3 つ以上ならば，最初の引数で渡されたプログラムから順番に処理を行う
285:         while (--argc) {
286: #ifdef MSC
287:             if ((fp = fopen(*++argv, "rb")) == NULL) {
288: #else
289:             if ((fp = fopen(*++argv, "r")) == NULL) {
290: #endif
291:                 fatal(*argv);
292:                 continue;
293:             }
294:             if ((length = strlen(*argv)) > namelen) } ファイル名の長さを調べる
295:                 namelen = length;                   }
296:             filename = *argv; ………ファイル名のセット
297:             root = xref(fp, root);
298:             fclose(fp);
```

```
299:             }
300:             break;
301:         }
302:         prnode(root);
303:         exit(0); ………プログラムの終了
304: }
```

[実行結果]

```
A>xref xref.c↵  ………このプログラム自身のクロスリファレンスをとってみる
DFLNAME     24 260
DFLNUMB     25 261
DFLSYMB     23 259
ENOMEM      94
EOF         67  75  80
EQUAL       261 263
FILE        54  84  85 218 259
GR  bol     45 145 161 233
            17
tail        31 118 207
token       60  77  83 159 161
tree         2
typedef     48  49  50
unsigned    17
void        89 203 228 238 246
while       80 196 279
xalloc      17 105 116 128 141
xfree       18
xlog10     178 198
xref         2   6 188 255 267 275 287

A>
```

リスト 10-1 XREF のプログラムリストと実行結果

## ■ プログラムの解説

このプログラムのなかで，とくに重要な「構造体を使った線形リストの管理方法」について以下で解説をしておきます．

＜30 行目〜33 行目＞

シンボルが存在する行番号を線形リストとして管理するための構造体を定義しています．この構造体は，自分自身の構造体へのポインタをメンバーとして持っており，これによって１つ前の構造体にリンクされています(図 10-1 を参照)．このように行番号を管理をすることによって，「何も書いていない行」などの無駄な行のためのエリアを取るようなことがなくなります．また，この構造体への実際の値の代入は，mkline 関数によって行われます．

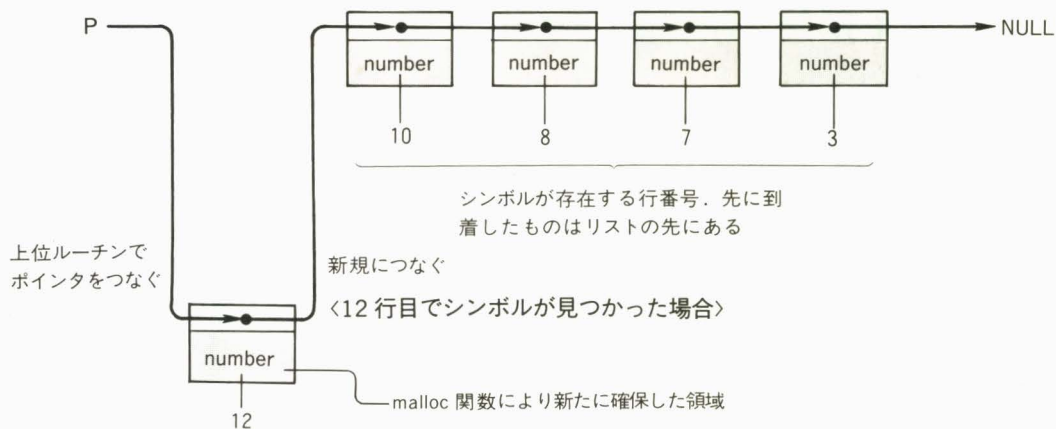



図10-1　_line 構造体による行番号の管理

**<35行目～39行目>**

この構造体は，複数のファイルが指定されたときに効率よくシンボルの管理を行うためのものです．構造体のメンバーnext は次の _link 構造体へのポインタを示し，head はそのシンボルのある行番号の構造体（_line 構造体）へのポインタとなっています（図10-2を参照）．実際には，mklink 関数がこの構造体の割り当てなどの管理をします．

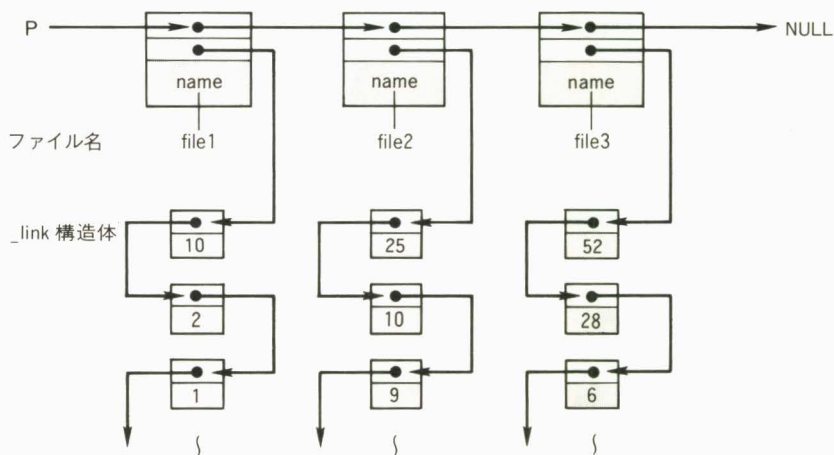



図10-2　_link 構造体によるファイル名の管理

**<41 行目〜46 行目>**

この構造体によって，最終的なシンボルの管理がツリー構造で行えます．構造体の実体は，mknode 関数によって作られ，mktree 関数によって left ポインタと right ポインタの値が決定します．（図 10-3 を参照）．

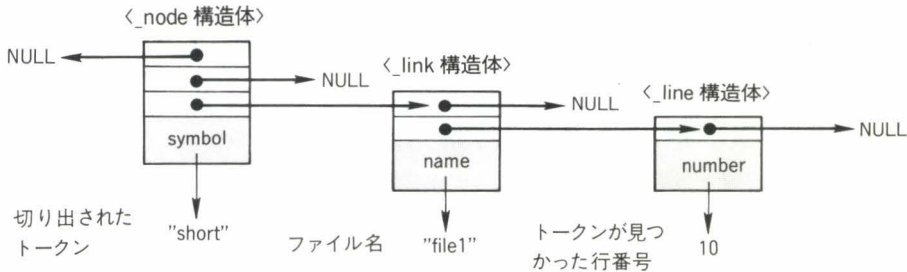



図 10-3 mknode 関数によって作られる構造体の実体

**<155 行目〜176 行目>**

mktree 関数は，シンボルが含まれる_node 構造体をツリー構造で管理するための関数です．まずシンボルがノードとして登録されているかどうかを再帰を使って検索し，各種の登録を行います．くわしくは，以下の図 10-4 を参照してください．

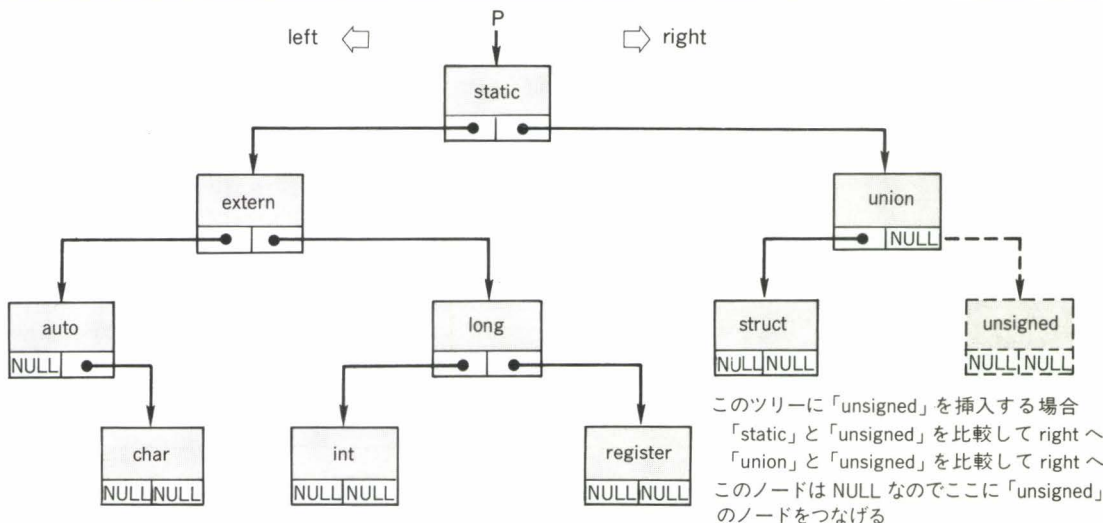



図 10-4 ツリー構造による_node 構造体の管理

# 10.2 LESS－逆スクロールが可能なmore

## ■ プログラムの概要

UNIX や MS-DOS には,「more」というユーティリティがあります．ご存じのように，このプログラムはテキストファイルのなかを見る場合，スクリーン１画面ごとに画面の動きを止めて，テキストの中身をより見やすくするためにできています．

しかし，すでに画面上からなくなってしまったテキストの部分は，もう一度 more を起動させてやらなければ見ることはできません．つまりテキストの前後を頻繁に参照しなければならない場合は，more よりもむしろスクリーンエディタを起動させるほうが便利です．しかしそれほどおおげさなものでなくても，簡単に逆スクロールできるプログラムとして作られたのがこの LESS です．

もとは USENET(UNIX ユーザーグループのネットワーク)で流れていたパブリックドメインのプログラムですが，パソコンに載せるにはあまりにも規模が大きすぎるため，機能を大幅に縮小し MS-DOS 用に新たに作り直しました．

コマンドはすべて＜数値＋１文字＞のコマンドで，１ページのスクロールと逆スクロール，そして，半ページのスクロールと逆スクロール，さらに，ファイルの最初と終わりに飛んで行くコマンドなどが用意されています．以下にコマンドの一覧表を示します(**表 10-1**)．

| コマンド | 機能 | コマンド | 機能 |
|---|---|---|---|
| <n>f<br><n>CTRL-F<br><n>スペース | <n> 画面先に進める | <n>u<br><n>CTRL-U | <n> 半画面後ろに戻す |
| <n>b<br><n>CTRL-B | <n> 画面後ろに戻す | <n>G | n 行へ移動．省略した場合はファイルの最後に移動 |
| <n>j<br><n>CTRL-J<br><n>CTRL-E<br><n>CTRL-M<br><n>リターン | <n> 行先に進める | <n>g | n 行へ移動．省略した場合はファイルの最初に移動 |
| <n>k<br><n>CTRL-K<br><n>CTRL-Y | <n> 行後ろに戻す | R | 現在の画面の書き直し(バッファをフラッシュ) |
| <n>d<br>CTRL-D | <n> 半画面先に進める | r<br>CTRL-R<br>CTRL-L | 現在の画面の書き直し |
| | | ＝<br>CTRL-G | 現在画面に表示されている最下位行の行番号とファイル名を表示 |
| | | q<br>CTRL-C | プログラムの終了 |

〈注意〉

- n は整数値を指定する．「G」,「g」コマンド以外では省略した場合は「1」となる
- コントロールキーと組み合わせたコマンドでは，大文字でも小文字でも指定できる

**表 10-1　LESS のコマンド一覧**

なお，ここで紹介するプログラムは，処理系(Microsoft C Compiler 特有の関数)とハードウェア(画面制御関係)に依存する部分がありますので注意してください．

## ■プログラムリスト

プログラムはLESSの本体である「LESS.C」とファイルのバッファ管理を行う「BUFF.C」そしてキーボードとCRTの入出力を扱う「TTY.C」の3つのファイルに分かれています．これらの部分をそれぞれ独立させたのは，処理系やハードウェアによって違いの出てくる部分をまとめることにより，他の処理系やハードウェアへの移植を容易に行えるようにするためです．

以下にそれぞれのファイルのプログラムリストを示します(**リスト10-2, リスト10-3, リスト10-4**)．

```
 1: /*
 2:     less.c - opposite of more
 3:         UNIX(LOCAL) like less command ultra super sub set
 4: */
 5:
 6: #include    <stdio.h>     } ヘッダファイルの取り込み
 7: #include    <ctype.h>
 8:
 9: #ifdef MSDOS
10: # include    <fcntl.h>    } MS-DOS 上の MS-C で必要なヘッダファイル
11: #endif
12:
13: #define    CNTRL(c)     ((c)-'@') ……コントロール文字のマクロ定義
14: #define    TABSTOP      8     ………タブの文字数
15: #define    NROW         23    ………1画面の行数．ファンクションキーを表示せずに使用する場合は，"24" とする
16: #define    NCOL         80    ………1行の文字数
17:
18: #define    w_clear()      fputs("¥033[2J", stdout)
19: #define    w_move(x, y)   printf("¥033[%d;%dH", y, x)    画面制御のための関数群の定義
20: #define    w_insert()     fputs("¥033[1L¥r", stdout)     エスケープシーケンスを使って
21: #define    w_delete()     fputs("¥033[1M¥r", stdout)     いるので移植の際には注意する
22: #define    w_normal()     fputs("¥033[0m", stdout)
23: #define    w_reverse()    fputs("¥033[7m", stdout)
24:
25: void    f_init();
26: char    *f_gets();
27: long    f_first();
28: long    f_last();
29:                          外部宣言が必要な関数群
30: void    t_init();
31: void    t_deinit();
32: char    t_getc();
33:
34: char    *progname;…………このプログラムの名前
35: FILE    *input;   ……………取り扱うファイルのFILE構造体へのポインタ
36: char    *filename;…………取り扱うファイル名
37: int ispipe;       ………………パイプ用の変数
```

```
38:
39: char     *expand(s)…………タブをスペースに展開する関数
40: char     *s;
41: {
42:     char    c, line[NCOL+1];
43:     int col;
44:
45:     for (col = Ø; col < NCOL; col++) {
46:         if ((c = *s++) == '¥Ø')
47:             break;
48:         if (c != '¥t') {
49:             line[col] = c;
50:             continue;
51:         }
52:         do
53:             line[col++] = ' ';
54:         while (col < NCOL && col % TABSTOP);
55:         col--;
56:     }
57:     line[col] = '¥Ø';
58:     fputs(line, stdout);
59:     return (s);
60: }
61:
62: long     forward(lineno, n) ………lineno 行から n 行分をファイルから読み込み表示する関数
63: long     lineno, n;
64: {
65:     long    i;
66:     char    *p;
67:
68:     for (i = Ø; i < n; i++) {
69:         if ((p = f_gets(lineno)) == NULL)
70:             break;       └────実際に 1 行読み出す関数(buff.c で記述)
71:         lineno++;
72:         expand(p);
73: #ifdef MSDOS
74:         fputs("¥r¥n", stdout);
75: #else
76:         fputs("¥n", stdout);
77: #endif
78:     }
79:     return (i);
80: }
81:
82: long     backward(lineno, n)………lineno 行から n 行分を逆方向に読み込み表示する関数
83: long     lineno, n;                なお、ここでは w_~ という関数でエスケープシーケンスを使っている
84: {
85:     long    i;
86:     char    *p;
87:
88:     w_move(1, 1);
89:     lineno -= NROW;
90:     for (i = Ø; i < n; i++) {
91:         if (lineno <= Ø || (p = f_gets(lineno)) == NULL)
92:             break;
```

```
 93:           lineno--;
 94:           w_insert();
 95:           expand(p);
 96:       }
 97:       w_move(1, NROW+1);
 98:       w_delete();
 99:       return (i);
100: }
101:
102: void     fill(n) ………表示する際に画面上のあまった行を「~」マークで埋める
103: long     n;
104: {
105: #ifdef MSDOS
106:      while (--n >= 0)
107:           fputs("~¥r¥n", stdout);
108: #else
109:      while (--n >= 0)
110:           fputs("~¥n", stdout);
111: #endif
112: }
113:
114: void     command() ………コマンドの解析と実行．先のforward関数とbackward関数の使われ方に注目してほしい
115: {
116:      long     lineno, val;
117:      int half;
118:      char     c;
119:
120:      half = NROW/2;
121:      w_clear();
122:      w_move(1, NROW+1);
123:      w_delete();
124:      lineno = forward(1L, (long)NROW);
125:      fill(NROW - lineno);
126:      if (filename != NULL) {
127:           w_reverse();
128:           fputs(filename, stdout);
129:           w_normal();
130:      } else
131:           putchar(':');
132:      for (;;) {
                    ┌──コンソールからの1文字入力用関数(tty.cで記述)
133:           val = 0;
134:           for (c = t_getc(); isdigit(c); c = t_getc()) {
135:                val *= 10;
136:                val += c - '0';
137:           }
138:           w_delete();
139:           switch (c) { ………コマンドによる分岐
140:           case 'f': case CNTRL('F'): case ' ': ………n画面先に進める
141:                if (val <= 0)
142:                     val = 1;
143:                lineno += forward(lineno+1, val*NROW);
144:                break;
145:           case 'b': case CNTRL('B'): ………n画面後ろに戻る
146:                if (val <= 0)
147:                     val = 1;
```

```
148:                lineno -= backward(lineno, val*NROW);
149:                break;
150:            case 'j': case CNTRL('J'): case CNTRL('E'): case CNTRL('M'): ………n 行先に進める
151:                if (val <= 0)
152:                    val = 1;
153:                lineno += forward(lineno+1, val);
154:                break;
155:            case 'k': case CNTRL('K'): case CNTRL('Y'): ……… n 行後ろに戻る
156:                if (val <= 0)
157:                    val = 1;
158:                lineno -= backward(lineno, val);
159:                break;
160:            case 'd': case CNTRL('D'): ……… n 半画面分先に進める
161:                if (val > 0 && val <= NROW)
162:                    half = val;
163:                else
164:                    val = half;
165:                lineno += forward(lineno+1, val);
166:                break;
167:            case 'u': case CNTRL('U'): ……… n 半画面分後に戻る
168:                if (val > 0 && val <= NROW)
169:                    half = val;
170:                else
171:                    val = half;
172:                lineno -= backward(lineno, val);
173:                break;
174:            case 'G': ………ファイルの最後へ移動
175:                if (val <= 0) {
176:                    lineno = f_last();
177:                    w_clear();
178:                    val = backward(lineno+NROW, (long)NROW);
179:                    w_move(1, (int)val+1);
180:                    fill(NROW - val);
181:                    break;
182:                }
183:                /* through down */
184:            case 'g': ………ファイルの最初に移動
185:                if (val < NROW)
186:                    val = NROW;
187:                if ((lineno = f_first()) > val)
188:                    val = lineno + NROW - 1;
189:                if (f_gets(val) == NULL)
190:                    val = f_last();
191:                lineno = val;
192:                goto redraw;
193:            case 'R': ………画面のリドロー
194:                if (ispipe)
195:                    goto error;
196:                f_init();
197:                fseek(input, 0L, 0);
198:                goto redraw;
199:            case 'r': case CNTRL('R'): case CNTRL('L'):
200:            redraw:
201:                if ((val = lineno - NROW + 1) <= 0)
```

```
202:                     val = 1L;
203:                 fill(NROW - forward(val, (long)NROW));
204:                 break;
205:             case '=': case CNTRL('G'): ………ファイル名と行番号の表示
206:                 w_reverse();
207:                 printf("%ld", lineno);
208:                 if (filename != NULL)
209:                     printf(" %s", filename);
210:                 w_normal();
211:                 continue;
212:             case 'q': case CNTRL('C'): ………プログラムの終了
213:                 return;
214:             error: ………誤ったコマンドの処理
215:             default:
216:                 putchar('¥007');
217:                 break;
218:             }
219:             putchar(':');
220:     }
221: }
222:
223: void    usage() ………プログラムの使い方の表示
224: {
225:     fprintf(stderr, "usage: %s [file]¥n", progname);
226: }
227:
228: void    main(argc, argv) ………メイン関数
229: int argc;
230: char    **argv;
231: {
232:     FILE    *fopen();
233:
234: #ifdef MSDOS
235:     progname = "less";
236: #else
237:     progname = argv[0];
238: #endif        ┌─ファイルハンドルが文字デバイス(コンソール,プリンタなど)かどうかを調べる
239:     if (argc == 1 && isatty(fileno(stdin))) { ………入力ファイルが間違っていないかどうか
240:         usage();                                    を調べる
241:         exit(0);                └─ファイルハンドルを返す
242:     }
243:     if (!isatty(fileno(stdout))) {
244:         fprintf(stderr, "%s: stdout must be a tty.¥n", progname);
245:         exit(1);
246:     }
247:     setbuf(stdout, (char *)NULL);
248:     setbuf(stderr, (char *)NULL);
249: #ifdef MSDOS
250:     setmode(fileno(stdout), O_BINARY); ………標準出力をバイナリモードにセット
251: #endif
252:     t_init(); ………標準入力の初期化(tty.c で記述)
253:     switch (argc) {
254:     case 1: ………ファイル名の指定がない場合
255:         input = stdin;
```

```
256:            filename = NULL;     ┐
257:            ispipe = 1;          │ パイプとして処理する
258:            command();           │
259:            break;               ┘
260:        case 2: ………ファイル名の指定がある場合
261:            if ((input = fopen(argv[1], "r")) == NULL) {
262:                perror(argv[1]);
263:                onintr(Ø);             MS-DOS 2.XX ではデバイスファイル(CON, AUX など)を指定すると
264:            }                          暴走することがある
265:            filename = argv[1];
266:            ispipe = isatty(fileno(input));……… デバイスをファイル名として指定したときはエラー
267:            if (ispipe) {                                                  (MS-DOS 3.1)
268:                fprintf(stderr, "*** %s is a device ***¥n", argv[1]);
269:                onintr(Ø);
270:            }
271:            command();
272:            fclose(input);
273:            break;
274:        default: ………その他の不正な入力は、使い方を表示してプログラムから抜ける
275:            usage();
276:            break;
277:        }
278:        t_deinit(); ………すべての状態を元に戻しプログラムを終了させる(tty.c で記述)
279:        exit(Ø);
280: }
```

リスト 10-2 LESS.C のプログラムリスト

```
  1: /*
  2:         buff.c - a parts of less
  3:             UNIX(LOCAL) like less command ultra super sub set
  4: */
  5:
  6: #include    <stdio.h>
  7:
  8: #ifdef MSC
  9: # include   <errno.h>  ┐ MS-C 独自のヘッダファイル(エラーナンバーが定義されている)
 10: #endif                 ┘
 11:
 12: #define     xalloc(t, n)    (t*)malloc((unsigned)sizeof(t)*(n)) ┐ malloc と free 関数を
 13: #define     xfree(p)        free((char *)p)                      ┘ 使いやすくするための
 14:                                                                    マクロ定義
 15: #define MAXLINE 256 ………扱う 1 行の最大長
 16:
 17: struct _buff { ………行バッファを管理する構造体
 18:     struct _buff *prev;……前の行の構造体へのポインタ
 19:     struct _buff *next;……次の行の構造体へのポインタ
 20:     long    number;   ………行番号
 21:     char    *string;  ………実際の行に入っている文字列のポインタ
 22: };
 23:
```

```
24: typedef struct _buff buff; ………この構造体を typedef しておく
25:
26: char    *malloc(); ………malloc 関数の使用宣言
27:
28: extern int      errno; ………エラーナンバーがセットされるグローバル変数 ┐
29: extern char    *progname; …このプログラムの名前                      ├ 実体は「less.c」にある
30: extern FILE    *input; ………入力ファイルのFILE 構造体へのポインタ      ┘
31:
32: static buff *current = NULL; ………現在の行構造体へのポインタ
33: static buff base; ………最初の行構造体
34:
35: void    f_init() ………バッファを管理するための定数の初期化関数
36: {
37:     buff    *p, *q;
38:
39:     if (current != NULL) { ………現在保持している行があればそれを解放する
40:         for (p = base.next; p != &base; p = q) {
41:             q = p->next;
42:             xfree(p->string);
43:             xfree(p);
44:         }
45:         current = NULL; ………保持している行がないように初期化
46:     }
47: }
48:
49: static void nocore() ………領域が確保できなかった場合に処理を終了する関数
50: {
51: #ifdef MSC
52:     errno = ENOMEM;
53: #endif
54:     perror(progname);
55:     onintr(0);
56: }
57:
58: static int frbuff() ………使わなくなった行バッファと行構造体のエリアを解放する関数
59: {
60:     buff    *p;
61:
62:     if ((p = base.next) == &base) ………base 構造体しかない場合はエラー
63:         return (-1);
64:     p->next->prev = p->prev; ┐
65:     p->prev->next = p->next; ┘ 行構造体のポインタの再設定
66:     xfree(p->string); ………行文字列の解放
67:     xfree(p); ………行構造体自身の解放
68:     return (0);
69: }
70:
71: static buff *mkbuff() ………新しい行をファイルから読んで行構造体のチェインを更新する
72: {
73:     char    *s, line[MAXLINE], *fgets();
74:     buff    *p;
75:     int last;
76:
```

```
 77:     if (fgets(line, MAXLINE, input) == NULL)
 78:         return (NULL);
 79:     for (last = strlen(line); last > 0; last--)
 80:         if (line[last-1] != '¥n')
 81:             break;
 82:     line[last++] = '¥0';
 83:     while ((p = xalloc(buff, 1)) == NULL)
 84:         if (frbuff())
 85:             nocore();
 86:     while ((s = xalloc(char, last)) == NULL)
 87:         if (frbuff())
 88:             nocore();
 89:     strcpy(s, line);
 90:     p->string = s;
 91:     p->number = ++base.number;
 92:     p->prev = base.prev;
 93:     p->next = &base;
 94:     p->prev->next = base.prev = p;
 95:     errno = 0;
 96:     return (p);
 97: }
 98:
 99: static buff *select(lineno)
100: long    lineno;
101: {
102:     if (current == NULL) {
103:         current = base.prev = base.next = &base;
104:         base.number = 0L;
105:         base.string = NULL;
106:     }
107:     if (current->number > lineno) {
108:         while ((current = current->prev) != &base)
109:             if (current->number == lineno)
110:                 return (current);
111:     } else {
112:         for (; current != &base; current = current->next)
113:             if (current->number == lineno)
114:                 return (current);
115:         while ((current = mkbuff()) != NULL)
116:             if (current->number == lineno)
117:                 return (current);
118:         current = base.prev;
119:     }
120:     return (NULL);
121: }
122:
123: char    *f_gets(lineno)
124: long    lineno;
125: {
126:     buff    *p;
127:
128:     if ((p = select(lineno)) == NULL)
129:         return (NULL);
130:     return (p->string);
131: }
```

エラーまたはEOFの場合，NULLを返す（77～78行）

1行の終わりの処理（79～82行）

もし確保できなければリストを1つ解放する（84～85行）

新しい行構造体の確保（83～85行）

新しい行バッファの確保（86～88行）

新しい行構造体のメンバーをそれぞれセットする（90～96行）

ポインタのチェインを更新（94行）

行番号で指定された行を取ってくる．結果は構造体に入るので，この関数ではそのポインタを返す（99行）

リストが存在しない場合，base構造体を次のように初期化（103～105行）

自分自身のアドレス（103行）



前方へサーチ（108～110行）

後方へサーチ（112～114行）

保持しているリストにないときはファイルから読み込む（115行）

読み込んだ行が目的行ならばリターン（116～117行）

これまで解説した管理方法を用いたファイルからの1行入力関数．標準関数のfgetsに当たる（123行）

行を取り込み，文字列のポインタを返す（128～130行）

```
132:
133: long    f_first() ……… 現在保持している最初の行構造体の行番号を返す
134: {
135:
136:     return (base.next->number);
137: }
138:
139: long    f_last() ……… ファイルの最後の行構造体の行番号を返す
140: {
141:     while (mkbuff() != NULL)
142:         ;
143:     current = base.prev;
144:     return (base.number);
145: }
```

リスト 10-3　BUFF.C のプログラムリスト

```
  1: /*
  2:     tty.c - a parts of less
  3:         UNIX(LOCAL) like less command ultra super sub set
  4: */
  5:
  6: #include    <stdio.h>
  7: #include    <errno.h>
  8: #include    <signal.h>
  9:
 10: #ifdef MSDOS
 11: # include    <fcntl.h>
 12: # include    <dos.h>
 13: #else
 14: # include    <sys/types.h>
 15: # include    <sys/ioctl.h>
 16: #endif
 17:
 18: #ifdef MSDOS
 19: # define    ISDEV     0x80  ┐
 20: # define    MASK      0x9f  │ MS-DOS のシステムコールで使われる定数ビットの定義
 21: # define    EOFBIT    0x40  │
 22: # define    RAWBIT    0x20  ┘
 23: #endif
 24:
 25: extern int errno; ………エラーナンバーがセットされるグローバル変数
 26:
 27: static int tty; ………コンソールが割り当てられているファイルのファイルハンドル
 28:
 29: int raw_mode(on) ………コンソールの入力を raw モードで行う用意をする MS-DOS だけに有効な関数
 30: int on;               くわしくはMS-DOS のシステムコールの解説を参照すること
 31: {
 32: #ifdef MSDOS
 33:     static unsigned save_term;
 34:     static unsigned save_intr;
```

```
35:     union REGS inregs, outregs;
36:
37:     if (on) {
38:         inregs.x.ax = 0x4400;  ………デバイス情報の取得
39:         inregs.x.bx = tty;
40:         intdos(&inregs, &outregs);
41:         if (outregs.x.cflag || !(outregs.h.dl & ISDEV))  ………デバイスでない場合はエラー
42:             return (-1);
43:         save_term = outregs.h.dl & MASK | EOFBIT;  ………現在の環境などを退避
44:         inregs.x.dx = save_term | RAWBIT;
45:     } else
46:         inregs.x.dx = save_term;  ………もとの環境を復帰
47:     inregs.x.ax = 0x4401;  ………デバイス情報の設定
48:     inregs.x.bx = tty;
49:     intdos(&inregs, &outregs);
50:     if (outregs.x.cflag)
51:         return (-1);
52:     if (on) {
53:         inregs.x.ax = 0x3300;  ………ブレークチェックの取得
54:         intdos(&inregs, &outregs);
55:         if (outregs.x.cflag)
56:             return (-1);
57:         save_intr = outregs.h.dl;  ………………環境を退避
58:         inregs.h.dl = 1;  ………………ブレークチェックを ON
59:     } else
60:         inregs.h.dl = save_intr;  ………………環境を復帰
61:     inregs.x.ax = 0x3301;
62:     intdos(&inregs, &outregs);  ………………ブレークチェックの設定
63:     if (outregs.x.cflag)
64:         return (-1);
65:     return (0);
66: #else                                                  ┐
67:     static struct sgttyb save_term;                    │
68:     struct sgttyb ttybuf;                              │
69:                                                        │
70:     if (on) {                                          │
71:         if (ioctl(tty, TIOCGETP, &ttybuf))             │
72:             return (-1);                               │
73:         save_term = ttybuf;      /* X3J11 */           │ UNIX 4.2BSD 用
74:         ttybuf.sg_flags |= CBREAK;                     │
75:         ttybuf.sg_flags &= ~(ECHO|XTABS);              │
76:     } else                                             │
77:         ttybuf = save_term;      /* X3J11 */           │
78:     if (ioctl(tty, TIOCSETN, &ttybuf))                 │
79:         return (-1);                                   │
80:     return (0);                                        ┘
81: #endif
82: }
83:
84: int onintr(sig)  ………ディスクアクセス中に CTRL-C による中断が行われた場合の処理
85: int sig;
86: {
87:     signal(SIGINT, SIG_IGN);
88:     raw_mode(0);
```

```
 89:     exit(2);
 9Ø: }
 91:
 92: void    t_init() ……… コンソールから 1 文字の即時入力
 93: {
 94:     signal(SIGINT, SIG_IGN);
 95: #ifdef MSDOS
 96: /* if ((tty = open("con", O_RDONLY|O_BINARY)) < Ø || raw_mode(1)) { */
 97:     tty = fileno(stderr);
 98: #else
 99:     tty = fileno(stderr);
1ØØ: #endif
1Ø1:     if (!isatty(tty) || raw_mode(1)) {
1Ø2:         errno = ENOTTY;
1Ø3:         perror("stderr");
1Ø4:         exit(2);
1Ø5:     }
1Ø6:     signal(SIGINT, onintr);
1Ø7: }
1Ø8:
1Ø9: void    t_deinit()
11Ø: {
111:     raw_mode(Ø);
112: }
113:
114: char    t_getc() ………コンソールI/O の最終処理
115: {
116:     char    c;
117:
118: #ifdef MSDOS
119:     while (read(tty, &c, sizeof(char)) != sizeof(char))
12Ø:         if (eof(tty))
121:             onintr(Ø);
122: #else
123:     while (read(tty, &c, sizeof(char)) != sizeof(char))
124:         ;
125: #endif
126:     return (c);
127: }
```

リスト 10-4　TTY.C のプログラムリスト

## ■実行ファイルの作成

LESSの実行ファイルを作るためには，3つのファイルを分割してコンパイルし，そのオブジェクトファイルをリンクしなければなりません(「9.2 分割コンパイル」を参照のこと)．以下に実行ファイルを作るためのバッチファイルを示します(図10-5)[†]．

```
A>type cc.bat
MSC LESS.C,  LESS.OBJ  /Zd/Od/Gs/G0/W1/J;
MSC BUFF.C, BUFF.OBJ /Zd/Od/Gs/G0/W1/J;
MSC TTY.C,  TTY.OBJ  /Zd/Od/Gs/G0/W1/J;
LINK LESS+BUFF+TTY+\LIB\SSETARGV, LESS.EXE, NULL, EM+SLIBFP+SLIBC/NOD/STACK:4096;

A>
```

└ワイルドカードを展開するためのライブラリ
本書のLESSでは複数ファイルをサポートしていないが，ファイル名を省略して入力したい場合などで使用可能

図10-5　実行ファイルを作成するためのバッチファイル

ここではMicrosoft C Compilerを用いています．他の処理系に移植した場合は同様の手順でコンパイルとリンクを行ってください．なお，図10-5ではコンパイラに各種のオプションを付けていますが，この指定はなくてもかまいません．また，リンク時にはコマンド行のワイルドカードの処理を行うために，「~SETARGV.OBJ」というオブジェクトファイルを標準ライブラリに合わせてリンクしています．

```
/*
        less.c - opposite of more
                UNIX(LOCAL) like less command ultra super sub set
*/

#include <stdio.h>
#include <ctype.h>

#ifdef MSDOS
# include <fcntl.h>
#endif

#define CNTRL(c)        ((c)-'@')
#define TABSTOP         8
#define NROW            23
#define NCOL            80

#define w_clear()       fputs("¥033[2J", stdout)
#define w_move(x, y)    printf("¥033[%d;%dH", y, x)
#define w_insert()      fputs("¥033[1L¥r", stdout)
#define w_delete()      fputs("¥033[1M¥r", stdout)
#define w_normal()      fputs("¥033[0m", stdout)
#define w_reverse()     fputs("¥033[7m", stdout)
LESS.C
     C1    CU    CA    S1    SU      VOID   NWL   INS   REP    ^Z
```

A>less less.c として，LESSを起動した画面．あとは表10-1に示すコマンドを使うことでスクロールをコントロールする

図10-6　LESSの実行結果

† clコマンドを用いてコンパイルとリンクを行うこともできる

以上のバッチファイルを実行すると，「LESS.EXE」実行ファイルが作成されます．次の**図 10-6** にその実行結果を示します．

ここでLESSはパイプからの入力も受け付けるので，たとえば，

```
A>sort  <  less.c ¦ less
```

というように起動することもできます．

なおこのプログラムでは，1行の文字数が80文字を越えた場合には画面が乱れます，また，漢字の処理もきちんと行っていませんから各自で拡張してください．

## ■行バッファの管理

このプログラムでは各行の管理に，双方向の2つのリンクポインタを持った線形リストを用いています．以下の**図 10-7** にその構造を示します．

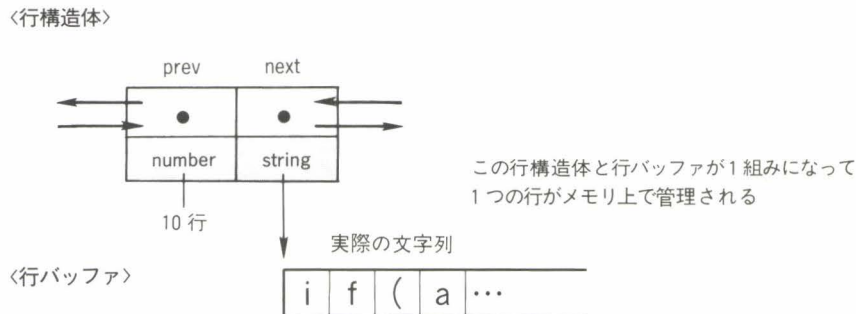



**図 10-7　行バッファを管理する構造体**

この構造体や行バッファの管理は，mkbuff関数やselect関数などで行っています．これらprevとnextの2つのポインタを持っていることによって，新規の挿入や解放がリンクポインタを掛け替えるだけで行うことができます．以下にその概念図を示します(**図 10-8**)．

また大きいファイルの場合，行のすべてをこのツリー構造のなかに入れることはできないので，これらのエリアの解放と再割り当ての管理も行います．このような管理の方法はかなり頻繁に行われる方法の1つですから，リストのコメントを参考にじっくりと解析してみてください．

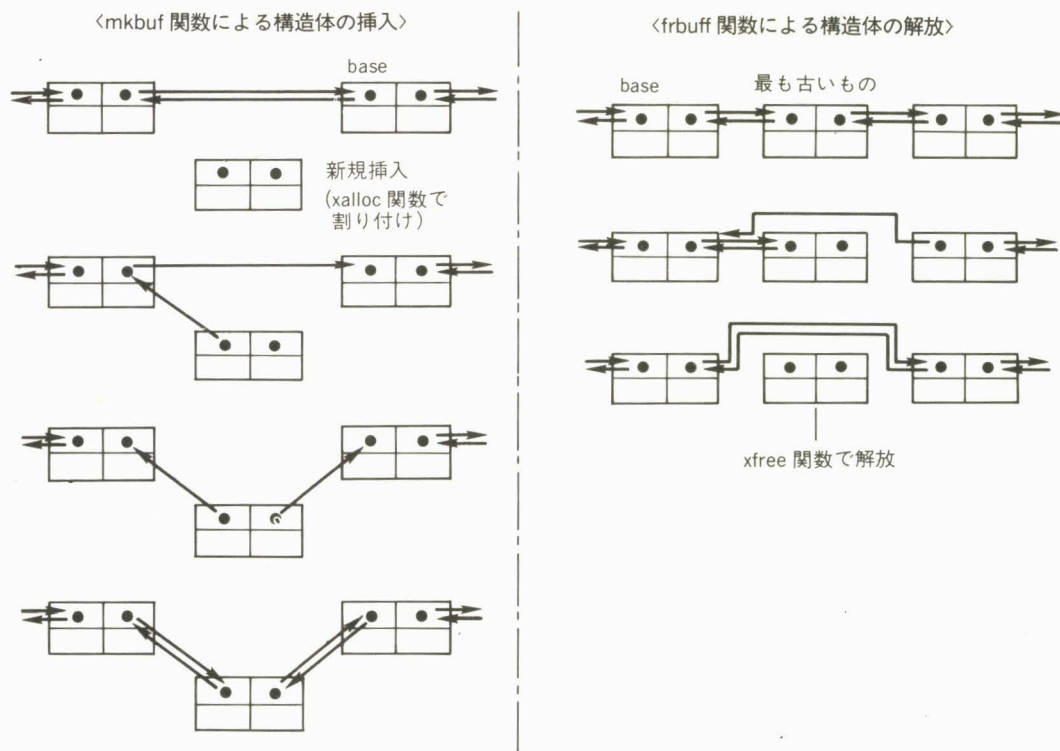



図 10-8　行構造体の新規の挿入と解放

## ■移植上の注意

Microsoft C Compiler 以外の処理系を使う場合は，TTY.C と BUFF.C，そして，LESS.C の一部を変更する必要があります．とくに，エスケープシーケンス（画面制御のための制御文字列）には気を付ける必要があります．もし，ここで使っているエスケープシーケンスが存在しない場合は，自分でそのエスケープシーケンスと同じような働きをする関数を作らなければなりません．この際，メモリ上に仮想スクリーンを想定しなければならない場合もでてきますので注意してください．

# APPENDIX

## ■ 各社の処理系一覧

本書でサポートした処理系を以下の表に紹介します．なお，価格，バージョン等は，昭和61年11月1日現在のものです．また，パソコン以外のC言語では，「UNIX 4.2BSD」上のC言語をサポートしています．各処理系のセットアップやコンパイルの方法などについては，前巻「入門C言語」のAPPENDIXでくわしく解説しているので参照してください．

昭和62年3月1日現在

| 処理系 | Ver. | OS | 発売元 | 住　所 | TEL | 価格 | 備　考 |
|---|---|---|---|---|---|---|---|
| Microsoft C Compiler | 3.0 | MS-DOS | マイクロソフト㈱ | 〒102　東京都千代田区三番町6-2 三番町弥生館 | 03(221)7074 | 98,000円 | |
| Lattice C | 3.1 | MS-DOS | ㈱ライフボート | 〒101　東京都千代田区神田錦町3～6 | 03(293)4711 | 98,000円 | |
| RUN/C | 2.0 | MS-DOS | | | | 33,000円 | |
| PM-MWC 86 | 2.15 | MS-DOS | パーソナルメディア㈱ | 〒141　東京都品川区西五反田2～12～19 五反田NNビル10F | 03(495)6241 | 189,000円<br>239,000円 | (フロッピーディスク版)<br>(ハードディスク対応版) |
| HI-TECH C | 3.05<br>3.04<br>3.05 | MS-DOS<br>CP/M-86<br>CP/M | ㈱サザンパシフィック | 〒220　神奈川県横浜市西区南幸2～16～20 三和横浜ビル | 045(314)9514 | 47,500円<br>47,500円<br>42,500円 | |
| DeSmet C | 2.5 | MS-DOS<br>CP/M-86 | ソフトウェア インターナショナル㈱ | 〒107　東京都港区南青山2～9～28 | 03(479)7151 | 46,000円 | |
| LSI C | 2.1 | CP/M | エル・エス・アイ ジャパン㈱ | 〒151　東京都渋谷区千駄ヶ谷1～8～14 | 03(478)0575 | 130,000円 | |

表A－1　C言語の処理系一覧表

## ■ プログラムの移植について

本書のプログラムはすべて「Microsoft C Compiler」で記述しており，「Microsoft C Compiler」と「UNIX 4.2BSD」上のC言語でチェックを行いました．ただし，標準的なC言語で記述しているので，多くのサンプルプログラムは上記のほとんどの処理系で変更せずに動作します．

第10章のプログラムなど，いくつかのプログラムについては，各処理系に合わせて移植を行う必要があります．他の処理系用に書かれたプログラムを移植する際に，とくに問題となるのは，プログラム中で使われているのと同じ標準関数が用意されているかどうかという点とその関数を使う上で必要なインクルードファイル(ヘッダファイル)は何かという点です．そこで，ここでは次ページ以降で2つの一覧表を紹介することにします．

## ■ 各処理系の標準関数一覧

以下で紹介する標準関数の一覧表は，「Microsoft C Compiler」と「Lattice C」の２つの処理系をもとに関数を選択し，機能別に分類しています．ただし処理系に固有の関数は排除し，できるだけ標準的なものを厳選しました．

この表では，移植のために各処理系に特定の関数が存在するかどうかの比較を目的としていますので，実際の使用にあたって必要な引数や返値(Return Value)を示していません．また，これらは処理系によって異なる場合があります．実際の移植にあたってはマニュアルを参照してください．

なお，この表は編集部の独自の調査によるものです．

表の見方

- ○は，その処理系に関数が存在することを示す
- 空白は，その処理系に関数が存在しないことを示す
- mは，その関数がインクルードファイルでマクロ定義されていることを示す
- 機能は，その関数の概略を示す．くわしくは各マニュアルを参照のこと

**標準入出力関数**

| 関数名 | 機能 | 4.2BSD | MS-C | Lattice | MWC | HI-TECH | DeSmet | RUN/C | LSI C |
|---|---|---|---|---|---|---|---|---|---|
| fgetc | ファイルから１文字読み込む | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| fgetchar | 標準入力から１文字読み込む | | ○ | ○ | | | | | |
| fgets | ファイルから１行読み込む | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| fprintf | ファイルに対しての書式付き出力 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ※1 |
| fputc | ファイルに１文字書き出す | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| fputchar | 標準出力に１文字書き出す | | ○ | ○ | | | | | |
| fputs | ファイルに１行書き出す | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| fread | ファイルからの読み込み | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| fscanf | ファイルからの書式付き入力 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ※1 |
| fwrite | ファイルへの書き出し | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| getc | ファイルから１文字読み込む | ○m | ○m | ○m | ○m | ○m | ○ | ○ | ○m |
| getchar | 標準入力から１文字読み込む | ○m | ○m | ○m | ○m | ○m | ○ | ○ | ○m |
| gets | ファイルから１行読み込む | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| getw | ファイルから１ワード読み込む | ○ | ○ | | ○m | ○ | ○ | | ○ |
| printf | 書式付き出力 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ※1 |
| putc | ファイルへ１文字書き出す | ○m | ○m | ○m | ○m | ○m | ○ | ○ | ○m |
| putchar | 標準出力へ１文字書き出す | ○m | ○m | ○m | ○m | ○m | ○ | ○ | ○m |
| puts | 標準出力へ文字列を書き出す | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| putw | ファイルへ１ワード書き出す | ○ | ○ | | ○m | ○ | ○ | | ○ |

※１…… 浮動小数点を扱う場合は，「fp_＋関数名」という関数を使う

| 関数名 | 機能 | 4.2BSD | MS-C | Lattice | MWC | HI-TECH | DeSmet | RUN/C | LSI C |
|---|---|---|---|---|---|---|---|---|---|
| scanf | 書式付き入力 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ※1 |
| sprintf | メモリへの書式付き出力 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ※1 |
| sscanf | メモリからの書式付き入力 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ※1 |
| ungetc | ファイルに1文字を戻す | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| clearerr | ファイルのエラーをリセット | ○ | ○ | ○m | ○m | ○ | | | ○ |
| fclose | ファイルをクローズする | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| fcloseall | すべてのファイルをクローズする | | ○ | ○ | | | | | |
| fdopen | ファイル・ハンドルを指定してファイルを開く | ○ | ○ | ○ | ○ | ○ | | | |
| feof | ファイルがEOFになったかどうかの判定 | ○m | ○m | ○m | ○m | ○m | | ○ | ○ |
| ferror | ファイルの読み書きのエラーの判定 | ○m | ○m | ○m | ○m | ○m | | ○ | ○ |
| fflush | ファイル・バッファのフラッシュ | ○ | ○ | ○m | ○ | ○ | ○ | ○ | ○ |
| fileno | ファイル・ハンドルを得る | ○m | ○m | ○m | ○m | ○m | | | |
| flushall | すべてのファイル・バッファをフラッシュする | | ○ | ○ | | | | | |
| fmode | ファイルの状態を変更する | | | ○ | | | | | |
| fopen | ファイルをオープン | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| fopene | 拡張機能を用いてファイルをオープン | | | ○ | | | | | |
| freopen | ファイルを再びオープン | ○ | ○ | ○ | ○ | ○ | | ○ | ○ |
| fseek | ファイルポインタの移動 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| ftell | ファイルポインタの位置を得る | ○ | ○ | ○ | ○ | ○ | | ○ | ○ |
| rewind | ファイルポインタを先頭へ戻す | ○ | ○ | ○m | ○ | ○ | ○ | ○ | ○ |
| setbuf | バッファ付き入出力の設定 | ○ | ○ | ○ | ○ | ○ | | | |
| exit | プログラムの終了 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| _exit | プログラムの終了 | | ○ | ○ | | | | | |
| **低水準入出力関数** | | | | | | | | | |
| chsize | ファイルの大きさを変更する | | ○ | | | | | | |
| close | ファイルをクローズ | ○ | ○ | ○ | ○ | ○ | | | ○ |
| creat | ファイルの作成 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| dup | ファイル・ハンドルの二重化 | ○ | ○ | ○ | ○ | ○ | | | |
| dup2 | ファイル・ハンドルの強制二重化 | ○ | ○ | ○ | ○ | ○ | | | |
| eof | ファイルのEOFの判定 | | ○ | | | | | | |
| filelength | ファイルの長さを得る | | ○ | | | | | | |
| getfc | ファイルの特性を得る | | | ○ | | | | | |
| iomode | ファイルの入出力モードを変更する | | | ○ | | | | | |
| lockf | ファイルのロックまたはアンロック | | | ○ | | | | | |
| locking | ファイルのロックまたはアンロック | | ○ | ○ | | | | | |
| lseek | ファイルポインタの移動 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| open | ファイルをオープン | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| opene | 拡張機能を用いてファイルをオープン | | | ○ | | | | | |

※1 …… 浮動小数点を扱う場合は，「fp_＋関数名」という関数を使う

| 関数名 | 機能 | 4.2BSD | MS-C | Lattice | MWC | HI-TECH | DeSmet | RUN/C | LSI C |
|---|---|---|---|---|---|---|---|---|---|
| read | ファイルからの読み込み | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| rlock | ファイルのロック | | | ○ | | | | | |
| runlk | ファイルのアンロック | | | ○ | | | | | |
| setmode | ファイルの変換モードを設定する | | ○ | | | | | | |
| sopen | シェアリングするファイルをオープン | | ○ | ※2 | | | | | |
| tell | ファイルポインタの現在位置を得る | ○ | ○ | ○ | | | | | ○ |
| umask | ファイルのパーミッションの初期設定 | ○ | ○ | | | | | | |
| write | ファイルへ書き出す | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| **ファイル関数** | | | | | | | | | |
| access | ファイルアクセスができるかどうかを調べる | ○ | ○ | ○ | | | | | |
| chdir | ディレクトリの移動 | ○ | ○ | ○ | | ○ | | | |
| chgfa | ファイルの属性を変更する | | | ○ | | | | | |
| chmod | ファイルの状態を変更する | ○ | ○ | ○ | | | | | |
| fstat | ファイルの状況を得る | ○ | ○ | | | | | | |
| getcd | 現在のディレクトリ位置を得る | | | ○ | | | | | |
| getcwd | 現在のディレクトリ位置を得る | | ○ | ○ | | | | | |
| getfa | ファイルの属性を得る | | | ○ | | | | | |
| getft | ファイルの作成時刻を得る | | | ○ | | | | | |
| mkdir | ディレクトリの作成 | ○ | ○ | ○ | | ○ | | | |
| remove | ファイルを削除する | | | ○ | | ○ | | | ○ m |
| rename | ファイル名の変更 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| rmdir | ディレクトリの削除 | ○ | ○ | ○ | | ○ | | | |
| stat | ファイルの状況を得る | ○ | ○ | | | | | | |
| unlink | ファイルを削除する | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| utime | ファイル作成時刻を変更する | ○ | ○ | chgft | | | | | |
| **プロセス・シグナル関数** | | | | | | | | | |
| abort | プログラムの中断 | ○ | ○ | ○ | | ○ | | | |
| execl | | ○ | ○ | ○ | | | | | ○ |
| execle | | ○ | ○ | ○ | | | | | |
| execlp | プロセスの移動 | ○ | ○ | ○ | | | chain | | |
| execv | | ○ | ○ | ○ | | | | | ○ |
| execve | | ○ | ○ | ○ | | | | | |
| execvp | | ○ | ○ | ○ | | | | | |
| getpid | プロセス識別子を得る | ○ | ○ | ○ | | | | | |
| onbreak | break 処理の設定 | | | ○ | | | | | |
| onerror | エラー処理の設定 | | | ○ | | | | | |
| signal | シグナル割り込みの設定 | ○ | ○ | ○ | | ○ | | | |
| system | DOS のコマンドの実行 | ○ | ○ | ○ | ○ | ○ | | | |

※2…… open 関数で制御できる

| 関数名 | 機能 | 4.2BSD | MS-C | Lattice | MWC | HI-TECH | DeSmet | RUN/C | LSI C |
|---|---|---|---|---|---|---|---|---|---|
| fork | プロセスの実行 | fork | spawnl<br>spawnle<br>spawnlp<br>spawnv<br>spawnve<br>spawnvp | forkl<br>forkle<br>forklp<br>forklpe<br>forkv<br>forkve<br>forkvp<br>forkvpe | | spawnl<br>spawnle<br>spawnv<br>spawnve | exec | | |
| wait | プロセスの実行を待つ | ○ | | ○ | | | | | |
| **メモリ関数** | | | | | | | | | |
| allmem | メモリエリアの割り当て | | | ○ | | | | | |
| bldmem | nk バイト単位でメモリエリアを得る | | | ○ | | | | | |
| calloc | 指定したサイズの配列エリアの割り付け | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| chkml | メモリエリアの最大ブロックの大きさを得る | | | ○ | | | | | |
| free | メモリエリアの解放 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| getmem<br>getml | メモリエリアを割り当てる | | | ○ | | | | | |
| lsbrk | メモリの割り当て | | | ○ | | | | | |
| malloc | メモリエリアの割り当て | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| rbrk | メモリエリアをリセットする | | | ○ | | | | | |
| realloc | メモリエリアの再割り当て | ○ | ○ | ○ | ○ | ○ | ○ | | ○ |
| rlsmem<br>rlsml | メモリエリアの解放 | | | ○ | | | | | |
| rstmem | メモリエリアを初期化する | | | ○ | | | | | |
| sbrk | メモリの割り当て | ○ | ○ | ○ | ○ | ○ | | | ○ |
| sizmem | メモリエリアの大きさを得る | | | ○ | | | | | |
| memcopy | メモリ・データのコピー | | ○ | ○ | | | | | |
| memchr | 文字の検索 | | ○ | ○ | | | | | ○ |
| memcmp | メモリ・データの比較 | | ○ | ○ | | ○ | | | ○ |
| memcpy | メモリ・データのコピー | | ○ | ○ | | ○ | | | ○ m |
| memset | メモリにデータを書き出す | | ○ | ○ | ※3 | ○ | | | ○ m |
| movedata | データの移動 | | ○ | ○ | | | lmove | | |
| movmem | データの移動 | | | ○ | | ○ | move | ○ | ○ |
| peek | メモリ・データの読み込み | | | ※4 | ○ | | ○ | ○ | |
| poke | メモリへの書き出し | | | ※4 | ○ | | ○ | ○ | |
| repmem | メモリ内容の複写 | | | ○ | | | | | |
| setmem | メモリブロックへデータを設定する | | | ○ | | | setmem | | ○ |
| swmem | メモリブロックの置き換え | | | ○ | | | | | |

※3 …… 似た関数に_zero がある
※4 …… メモリの指定にセグメントとオフセットを用いる

| 関数名 | 機能 | 4.2BSD | MS-C | Lattice | MWC | HI-TECH | DeSmet | RUN/C | LSI C |
|---|---|---|---|---|---|---|---|---|---|
| **時間関数** | | | | | | | | | |
| asctime | 時刻を文字化して得る | ○ | ○ | ○ | | ○ | | | |
| ctime | 1970 年 1 月 1 日からの秒数を得る | ○ | ○ | ○ | | ○ | | | |
| ftime | 現在の時刻を得る | ○ | ○ | | | | | | |
| getclk | 時刻を得る | | | ○ | | | | | |
| gmtime | グリニッジ標準時刻を得る | ○ | ○ | ○ | | ○ | | | |
| localtime | 地域別の時刻を得る | ○ | ○ | ○ m | | ○ | ※ 5 | ※ 6 | |
| stpdate | 日付データを文字列に変換する | | | ○ | | | | | |
| stptime | 時刻データを文字列に変換する | | | ○ | | | | | |
| time | システムの時間を秒単位で得る | ○ | ○ | ○ | | ○ | | | |
| tzname | 時間名を得る | | | ○ | | | | | |
| tzset | 時間名を設定する | | ○ | ○ | | | | | |
| **文字評価・文字列関数** | | | | | | | | | |
| isalnum | 英数字かどうかを調べる | ○ m | ○ m | ○ m | ○ m | ○ m | ○ | ○ | ○ m |
| isalpha | 英字かどうかを調べる | ○ m | ○ m | ○ m | ○ m | ○ m | ○ | ○ | ○ m |
| isascii | ASCII 文字かどうかを調べる | ○ m | ○ m | ○ m | ○ m | ○ m | ○ | ○ | ○ m |
| iscntrl | コントロール文字かどうかを調べる | ○ m | ○ m | ○ m | ○ m | ○ m | ○ | ○ | ○ m |
| isdigit | 数字かどうかを調べる | ○ m | ○ m | ○ m | ○ m | ○ m | ○ | ○ | ○ m |
| isgraph | グラフィック文字かどうかを調べる | | ○ m | ○ m | | ○ m | | ○ | ○ m |
| islower | 英小文字かどうかを調べる | ○ m | ○ m | ○ m | ○ m | ○ m | ○ | ○ | ○ m |
| isprint | 表示可能な文字かどうかを調べる | ○ m | ○ m | ○ m | ○ m | ○ m | ○ | ○ | ○ m |
| ispunct | 区切り文字かどうかを調べる | ○ m | ○ m | ○ m | ○ m | ○ m | ○ | ○ | ○ m |
| isspace | 空白文字かどうかを調べる | ○ m | ○ m | ○ m | ○ m | ○ m | ○ | ○ | ○ m |
| isupper | 英大文字かどうかを調べる | ○ m | ○ m | ○ m | ○ m | ○ m | ○ | ○ | ○ m |
| isxdigit | 16 進文字かどうかを調べる | ○ m | ○ m | ○ m | | ○ m | | ○ | ○ m |
| toascii | ASCII 文字に変換する | ○ m | ○ m | ○ m | ○ m | ○ m | | | |
| tolower | 大文字を小文字に変換する | ○ m | ○ m | ○ m | ○ m | ○ m | ○ | ○ | ○ |
| toupper | 小文字を大文字に変換する | ○ m | ○ m | ○ m | ○ m | ○ m | ○ | ○ | ○ |
| _tolower | 大文字を小文字に変換する | | ○ m | | ○ m | | | | |
| _toupper | 小文字を大文字に変換する | | ○ m | | ○ m | | | | |
| strcat | 文字列の結合 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| strcmp | 文字列の比較 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| strcmpi | 文字列の比較 | | ○ | ○ m | | | | | |
| strcpy | 文字列の複写 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| strdup | 文字列の複製 | | ○ | ○ | | | | | |
| strins | 文字列の挿入 | | | ○ | | | ○ | | |
| strlen | 文字列の長さを得る | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |

※ 5 …… 似た関数に times, dates がある
※ 6 …… 似た関係に dosdate, dostime がある

| 関数名 | 機能 | 4.2BSD | MS-C | Lattice | MWC | HI-TECH | DeSmet | RUN/C | LSI C |
|---|---|---|---|---|---|---|---|---|---|
| strncat | 文字列に n 文字の文字列を加える | ○ | ○ | ○ | ○ | ○ | ○ |  | ○ |
| strncmp | 文字列を n 文字だけ比較する | ○ | ○ | ○ | ○ | ○ | ○ |  | ○ |
| strncpy | 文字列を n 文字複写する | ○ | ○ | ○ | ○ | ○ | ○ |  | ○ |
| strnset | 長さを指定して文字を書き込む |  | ○ | ○ |  |  |  |  |  |
| strset | 文字を書き込む |  | ○ | ○ |  |  |  |  |  |
| index | 文字を捜す | ○ |  |  | ○ | ○ | ○ |  |  |
| rindex | 文字を捜す | ○ |  |  | ○ | ○ | ○ |  |  |
| strchr | 文字を捜す |  | ○ | ○ |  | ○ |  |  | ○ |
| strcspn | 文字列の文字構成の比較 |  | ○ | ○ |  |  |  |  | ○ |
| strpbrk | 区切り文字を捜す |  | ○ | ○ |  |  |  |  | ○ |
| strrchr | 前方に向かって文字を捜す |  | ○ | ○ |  | ○ |  |  | ○ |
| strspn | 文字列の文字構成の比較 |  | ○ | ○ |  |  |  |  | ○ |
| strtok | トークン文字を得る |  | ○ | ○ |  |  |  |  | ○ |
| atof | 文字列を float 型に変換 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| atoi | 文字列を int 型に変換 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| atol | 文字列を long 型に変換 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| ecvt | float 型を文字列に変換 | ○ | ○ | ○ | ○ |  |  |  |  |
| fcvt | float 型を文字列に変換 | ○ | ○ | ○ | ○ |  |  |  |  |
| gcvt | float 型を文字列に変換 | ○ | ○ | ○ | ○ |  |  |  |  |
| itoa | int 型を文字列に変換 |  | ○ | ※7 |  |  |  |  |  |
| ltoa | long 型を文字列に変換 |  | ○ |  |  |  |  |  |  |
| ultoa | unsigned long 型を文字列に変換 |  | ○ |  |  |  |  |  |  |
| strlwr | 文字列を小文字に変換する |  | ○ | ○ |  |  |  |  |  |
| strrev | 文字列の並びを逆にする |  | ○ | ○ |  |  |  |  |  |
| strupr | 文字列を大文字にする |  | ○ | ○ |  |  |  |  |  |
| **算術関数** |  |  |  |  |  |  |  |  |  |
| abs | 絶対値を求める | ○ | ○m | ○m | ○ | ○ | ○ |  | ○ |
| acos | 余弦の逆関数 | ○ | ○ | ○ | ○ | ○ | ○ | ○ |  |
| asin | 正弦の逆関数 | ○ | ○ | ○ | ○ | ○ | ○ | ○ |  |
| atan | 正接の逆関数 | ○ | ○ | ○ | ○ | ○ | ○ | ○ |  |
| atan2 | 正接の逆関数 | ○ | ○ | ○ | ○ | ○ |  | ○ |  |
| cabs | 複素数の絶対値を求める | ○ | ○ |  | ○ |  |  |  |  |
| ceil | 引数に最も近い整数を得る | ○ | ○ | ○ | ○ | ○ | ○ | ○ |  |
| cos | 余弦関数 | ○ | ○ | ○ | ○ | ○ | ○ | ○ |  |
| cosh | ハイパボリック余弦関数 | ○ | ○ | ○ | ○ | ○ |  | ○ |  |
| exp | 指数関数を計算する | ○ | ○ | ○ | ○ | ○ | ○ | ○ |  |
| fabs | float 型の絶対値を得る | ○ | ○ | ○ | ○ | ○ | ○ | ○ |  |

※7 …… stci_d 関数（int 型を文字列に変換）／stcu_d 関数（unsigned 型を文字列に変換）
Lattice C の「is～」，「to～」関数は fctype.h をインクルードすればマクロではなく，関数となる

| 関数名 | 機能 | 4.2BSD | MS-C | Lattice | MWC | HI-TECH | DeSmet | RUN/C | LSI C |
|---|---|---|---|---|---|---|---|---|---|
| floor | 最も近い整数値を float 型で得る | ○ | ○ | ○ | ○ | ○ | ○ | ○ | |
| fmod | float 型の剰余を得る | | ○ | ○ | | | | | |
| frexp | 指数部と仮数部を分離する | ○ | ○ | ○ | ○ | ○ | ○ | | |
| hypot | 直角三角形の対角長を得る | ○ | ○ | | ○ | | | | |
| iabs | 整数の絶対値を求める | | | ○ | | | | | |
| labs | long 型の絶対値を得る | | ○ | ○ | | | | | |
| ldexp | 2 の X 乗を計算する | ○ | ○ | ○ | ○ | ○ | ○ | | |
| log | 自然対数を求める | ○ | ○ | ○ | ○ | ○ | ○ | ○ | |
| log10 | 10 を底とする対数を求める | ○ | ○ | ○ | ○ | ○ | ○ | ○ | |
| matherr | 演算エラー処理 | | ○ | ○ | | | | | |
| modf | 整数部と分数部に分離する | ○ | ○ | ○ | ○ | | ○ | | |
| pow | X の Y 乗を計算する | ○ | ○ | ○ | ○ | ○ | ○ | ○ | |
| rand | 乱数を発生する | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| sin | 正弦関数 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | |
| sinh | ハイパボリック正弦関数 | ○ | ○ | ○ | ○ | ○ | | ○ | |
| sqrt | 平方根 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | |
| srand | 乱数の初期値を設定する | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| tan | 正接関数 | ○ | ○ | ○ | ○ | ○ | ○ | ○ | |
| tanh | ハイパボリック正接関数 | ○ | ○ | ○ | ○ | ○ | | ○ | |
| **DOS インターフェイス関数** | | | | | | | | | |
| bdos | DOS のシステムコールの実行 | | ○ | ○ | | msdos | os | | ○ |
| bdosx | BDOS でセグメントを必要とする場合 | | | ○ | | | | | |
| dosexterr | 拡張エラーコードを得る | | ○ | | | ○ | | | |
| int86 | ソフトウェア割り込みの発生 | | ○ | ○ | | ○ | doint | | |
| int86x | ソフトウェア割り込みの発生 | | ○ | ○ | intcal | ○ | | | |
| intdos | MS-DOS のシステムコールを行う | | ○ | ○ | | ○ | | ○ | |
| intdosx | MS-DOS のシステムコールを行う | | ○ | ○ | | ○ | | ○ | |
| segread | セグメントを読み込む | | ○ | ○ | | ○ | ※8 | | |
| inp | I/O ポートからの入力値を得る | | ○ | ○ | inb | ○ | inb | ○ | ○ m |
| outp | I/O ポートへデータを出力する | | ○ | ○ | outb | ○ | outb | ○ | ○ m |
| FP_OFF | ポインタのオフセットを求める | | ○ m | ○ | ※9 | ○ | | | |
| FP_SEG | ポインタのセグメントを求める | | ○ m | ○ | ※9 | ○ | | | |
| cgets | コンソールから 1 行読み込む | | ○ | ○ | | ○ | | | |
| cprintf | コンソールへの書式付き出力 | | ○ | ○ | | | | | |
| cputs | 文字列をコンソールへ書き出す | | ○ | ○ | | ○ | | | |
| cscanf | 書式付きコンソール入力 | | ○ | ○ | | | | | |
| getch | コンソールから 1 文字読み込む | | ○ | ○ | | ○ | ci | ○ | |

※ 8 …… 似た関数に_showcs, _showds, _showsp がある
※ 9 …… 似た関数に ptoreg がある

| 関数名 | 機能 | 4.2BSD | MS-C | Lattice | MWC | HI-TECH | DeSmet | RUN/C | LSI C |
|---|---|---|---|---|---|---|---|---|---|
| getche | コンソールから１文字読み込む | | ○ | ○ | getcnb | ○ | | | |
| putch | コンソールへ１文字書き出す | | ○ | ○ | putcnb | ○ | co | ○ | |
| ungetch | コンソールへ１文字戻す | | ○ | ○ | | ○ | | ○ | ○ m |
| kbhit | キーボード入力を調べる | | ○ | ○ | | ○ | | inkey | ○ |
| **その他の関数** | | | | | | | | | |
| assert | プログラムの検証 | ○ | ○ m | ○ m | ○ m | ○ m | | | |
| getenv | 環境変数を得る | ○ | ○ | ○ | ○ | ○ | | | |
| putenv | 環境変数の設定 | | ○ | ○ | | | | | |
| longjmp | グローバルなジャンプ | ○ | ○ | ○ | ○ | ○ | ○ | | ○ |
| setjmp | グローバルなジャンプの設定 | ○ | ○ | ○ | ○ | ○ | ○ | | ○ |
| isatty | ファイルがデバイスがどうかを調べる | ○ | ○ | ○ | | ○ | | | |
| perror | エラーメッセージの出力 | ○ | ○ | ○ | | ○ | | | |
| poserr | MS-DOSのエラーメッセージの出力 | | | ○ | | | | | |
| swab | バイト単位の入れ替え | ○ | ○ | | ○ | | | | |
| bserch | ソートされた配列のバイナリサーチ | | ○ | | | | | | |
| qsort | クイックソート | ○ | ○ | ○ | ○ | ○ | ○ | | ○ |

表A－2　各処理系の標準関数一覧表

## ■各処理系のインクルードファイル一覧

前述の標準関数を使う上で必要なインクルードファイル(ヘッダファイル)の一覧表を以下に示します．インクルードの方法やそのファイルの中身については，第9章でくわしく解説していますから参照してください．

表の見方

- ○は，関数を使う上でファイルをインクルードする必要がないことを示す
- 空白は，その処理系に関数が存在しないことを示す
- ▽は，同じ働きをする関数があることを示す（標準関数の一覧表を参照）
- ◇は，似かよった関数があることを示す（標準関数の一覧表を参照）

**標準入出力関数**

| 関数名 | 4.2BSD | MS-C | Lattice | MWC | HI-TECH | DeSmet | RUN/C | LSI C |
|---|---|---|---|---|---|---|---|---|
| fgetc | stdio.h | stdio.h | stdio.h | ○ | ○ | ○ | ○ | stdio.h |
| fgetchar | | stdio.h | stdio.h | | | | | |
| fgets | stdio.h | stdio.h | stdio.h | stdio.h | stdio.h | ○ | ○ | stdio.h |
| fprintf | stdio.h | stdio.h | stdio.h | ○ | ○ | ○ | ○ | stdio.h |
| fputc | stdio.h | stdio.h | stdio.h | ○ | ○ | ○ | ○ | stdio.h |
| fputchar | | stdio.h | stdio.h | | | | | |
| fputs | stdio.h | stdio.h | stdio.h | stdio.h | ○ | ○ | ○ | stdio.h |
| fread | stdio.h | stdio.h | stdio.h | ○ | ○ | ○ | ○ | stdio.h |
| fscanf | stdio.h | stdio.h | stdio.h | ○ | ○ | ○ | ○ | stdio.h |
| fwrite | stdio.h | stdio.h | stdio.h | ○ | ○ | ○ | ○ | stdio.h |
| getc | stdio.h | stdio.h | stdio.h | stdio.h | stdio.h | ○ | ○ | stdio.h |
| getchar | stdio.h | stdio.h | stdio.h | stdio.h | stdio.h | ○ | ○ | stdio.h |
| gets | stdio.h | stdio.h | stdio.h | stdio.h | ○ | ○ | ○ | stdio.h |
| getw | stdio.h | stdio.h | | stdio.h | ○ | ○ | | stdio.h |
| printf | stdio.h | stdio.h | stdio.h | ○ | ○ | ○ | ○ | stdio.h |
| putc | stdio.h | stdio.h | stdio.h | stdio.h | stdio.h | ○ | ○ | stdio.h |
| putchar | stdio.h | stdio.h | stdio.h | stdio.h | stdio.h | ○ | ○ | stdio.h |
| puts | stdio.h | stdio.h | stdio.h | stdio.h | ○ | ○ | ○ | stdio.h |
| putw | stdio.h | stdio.h | | stdio.h | ○ | ○ | | stdio.h |
| scanf | stdio.h | stdio.h | stdio.h | ○ | ○ | ○ | ○ | stdio.h |
| sprintf | stdio.h | stdio.h | stdio.h | ○ | ○ | ○ | ○ | stdio.h |
| sscanf | stdio.h | stdio.h | stdio.h | ○ | ○ | ○ | ○ | stdio.h |
| ungetc | stdio.h | stdio.h | stdio.h | ○ | ○ | ○ | ○ | stdio.h |
| clearerr | stdio.h | stdio.h | stdio.h | stdio.h | ○ | | | stdio.h |
| fclose | stdio.h | stdio.h | stdio.h | ○ | ○ | ○ | ○ | stdio.h |
| fcloseall | | stdio.h | stdio.h | | | | | |
| fdopen | stdio.h | stdio.h | stdio.h | stdio.h | stdio.h | | | |
| feof | stdio.h | stdio.h | stdio.h | stdio.h | stdio.h | | ○ | stdio.h |
| ferror | stdio.h | stdio.h | stdio.h | stdio.h | stdio.h | | ○ | stdio.h |
| fflush | stdio.h | stdio.h | stdio.h | ○ | ○ | ○ | ○ | stdio.h |
| fileno | stdio.h | stdio.h | stdio.h | stdio.h | stdio.h | | | |
| flushall | | stdio.h | stdio.h | | | | | |
| fmode | | | stdio.h | | | | | |
| fopen | stdio.h | stdio.h | stdio.h | stdio.h | stdio.h | ○ | ○ | stdio.h |
| fopene | | | stdio.h | | | | | |
| freopen | stdio.h | stdio.h | stdio.h | stdio.h | stdio.h | | ○ | stdio.h |
| fseek | stdio.h | stdio.h | stdio.h | ○ | ○ | ○ | ○ | stdio.h |
| ftell | stdio.h | stdio.h | stdio.h | stdio.h | stdio.h | | ○ | stdio.h |

| 関数名 | 4.2BSD | MS-C | Lattice | MWC | HI-TECH | DeSmet | RUN/C | LSI C |
|---|---|---|---|---|---|---|---|---|
| rewind | stdio.h | stdio.h | stdio.h | ○ | ○ | ○ | ○ | stdio.h |
| setbuf | stdio.h | stdio.h | stdio.h | stdio.h | ○ | | | |
| exit | ○ | process.h | stdlib.h | assert.h | ○ | ○ | ○ | stdio.h |
| _exit | process.h | stdlib.h | | | | | | |
| **低水準入出力関数** | | | | | | | | |
| chsize | | io.h | | | | | | |
| close | ○ | io.h | fcntl.h | ○ | ○ | | | stdio.h |
| creat | ○ | io.h | fcntl.h | ○ | ○ | ○ | ○ | stdio.h |
| dup | ○ | io.h | fcntl.h | ○ | ○ | | | |
| dup2 | ○ | io.h | fcntl.h | ○ | ○ | | | |
| eof | | io.h | | | | | | |
| filelength | | io.h | | | | | | |
| getfc | | | dos.h | | | | | |
| iomode | | | fcntl.h | | | | | |
| iockf | | | unlstd.h | | | | | |
| locking | | io.h | ○ | | | | | |
| lseek | ○ | io.h | fcntl.h | ○ | ○ | ○ | ○ | stdio.h |
| open | sys/file.h | io.h | fcntl.h | ○ | ○ | ○ | ○ | stdio.h |
| opene | | | ○ | | | | | |
| read | ○ | io.h | fcntl.h | ○ | ○ | ○ | ○ | stdio.h |
| rlock | | | dos.h | | | | | |
| runlk | | | dos.h | | | | | |
| setmode | | io.h | | | | | | |
| sopen | | io.h | ◇ | | | | | |
| tell | ○ | io.h | fcntl.h | | | | | stdio.h |
| umask | ○ | io.h | | | | | | |
| write | ○ | io.h | fcntl.h | ○ | ○ | ○ | ○ | stdio.h |
| **ファイル関数** | | | | | | | | |
| access | sys/file.h | io.h | stdio.h | | | | | |
| chdir | ○ | direct.h | stdio.h | | ○ | | | |
| chgfa | | | dos.h | | | | | |
| chmod | ○ | io.h | stdio.h | | | | | |
| fstat | sys/types.h<br>sys/stat.h | stat.h | | | | | | |
| getcd | | | dos.h | | | | | |
| getcwd | | direct.h | stdio.h | | | | | |
| getfa | | | dos.h | | | | | |
| getft | | | dos.h | | | | | |
| mkdir | ○ | direct.h | stdio.h | | ○ | | | |

| 関数名 | 4.2BSD | MS-C | Lattice | MWC | HI-TECH | DeSmet | RUN/C | LSI C |
|---|---|---|---|---|---|---|---|---|
| remove | | | fcntl.h | | ○ | | | stdio.h |
| rename | ○ | io.h | stdio.h | ○ | ○ | ○ | ○ | stdio.h |
| rmdir | ○ | direct.h | stdio.h | | ○ | | | |
| stat | sys/types.h<br>sys/stat.h | stat.h | | | | | | |
| unlink | ○ | io.h | fcntl.h | ○ | ○ | ○ | ○ | stdio.h |
| utime | sys/types.h | | ◇ | | | | | |
| **プロセス・シグナル関数** | | | | | | | | |
| abort | ○ | process.h | stdlib.h | | ○ | | | stdlib.h |
| execl<br>execle<br>execlp<br>execv<br>execve<br>execvp | ○<br>○<br>○<br>○<br>○<br>○ | process.h<br>process.h<br>process.h<br>process.h<br>process.h<br>process.h | stdlib.h<br>stdlib.h<br>stdlib.h<br>stdlib.h<br>stdlib.h<br>stdlib.h | | | chain | | stdlib.h |
| getpid | ○ | process.h | stdlib.h | | | | | |
| onbreak | | | dos.h | | | | | |
| onerror | | | dos.h | | | | | |
| signal | signal.h | signal.h | signal.h | | signal.h | | | |
| system | ○ | process.h | stdlib.h | | | | | |
| folk | ※1 | process.h<br>※2 | stdlib.h<br>※3 | | ※2 | ※4 | | |
| wait | sys/wait.h | | stdlib.h | | | | | |
| **メモリ関数** | | | | | | | | |
| allmem | | | stdlib.h | | | | | |
| bldmem | | | stdlib.h | | | | | |
| calloc | ○ | malloc.h | stdlib.h | ○ | ○ | ○ | ○ | stdlib.h |
| chkml | | | stdlib.h | | | | | |
| free | ○ | malloc.h | stdlib.h | | ○ | ○ | ○ | stdlib.h |
| getmem<br>getml | | | stdlib.h<br>stdlib.h | | | | | |
| lsbrk | | | stdlib.h | | | | | |
| malloc | ○ | malloc.h | stdlib.h | stdio.h | ○ | ○ | ○ | stdlib.h |
| rbrk | | | stdlib.h | | | | | |
| realloc | ○ | malloc.h | stdlib.h | ○ | ○ | ○ | | stdlib.h |
| malloc | ○ | malloc.h | stdlib.h | stdio.h | ○ | ○ | ○ | stdlib.h |
| rbrk | | | stdlib.h | | | | | |
| realloc | ○ | malloc.h | stdlib.h | ○ | ○ | ○ | | stdlib.h |
| rlsmem<br>rlsm | | | stdlib.h<br>stdlib.h | | | | | |

※1 …… fork 関数
※2 …… spawnl, spawny などの関数
※3 …… forkl, forkv などの関数
※4 …… exec 関数

| 関数名 | 4.2BSD | MS-C | Lattice | MWC | HI-TECH | DeSmet | RUN/C | LSI C |
|---|---|---|---|---|---|---|---|---|
| rstmem | | | stdlib.h | | | | | |
| sbrk | ○ | malloc.h | stdlib.h | stdio.h | ○ | | | stdlib.h |
| sizmem | | | stdlib.h | | | | | |
| memcopy | | memory.h | string.h | | | | | |
| memchr | | memory.h | string.h | | | | | string.h |
| memcmp | | memory.h | string.h | | ○ | | | string.h |
| memcpy | | memory.h | string.h | | ○ | | | string.h |
| memset | | memory.h | string.h | ◇ | ○ | | | string.h |
| movedata | | memory.h | dos.h | | | ▽ | | |
| movmem | | | string.h | | ○ | ▽ | ○ | string.h |
| peek | | | dos.h | ○ | | ▽ | ○ | |
| poke | | | dos.h | ○ | | ▽ | ○ | |
| repmem | | | string.h | | | | | |
| setmem | | | string.h | | | ▽ | | string.h |
| swmem | | | string.h | | | | | |
| **時間関数** | | | | | | | | |
| asctime | sys/time.h | time.h | time.h | | time.h | | | |
| ctime | sys/time.h | time.h | time.h | | time.h | | | |
| ftime | sys/types.h<br>sys/timeb.h | timeb.h | | | | | | |
| getclk | | | dos.h | | | | | |
| gmtime | sys/time.h | time.h | time.h | | time.h | | | |
| localtime | sys/time.h | time.h | time.h | | time.h | ◇ | ◇ | |
| stpdate | | | string.h | | | | | |
| stptime | | | string.h | | | | | |
| time | ○ | time.h | time.h | | time.h | | | |
| tzname | | | time.h | | | | | |
| tzset | | time.h | time.h | | | | | |
| **文字評価・文字列関数** | | | | | | | | |
| isalnum | ctype.h | ctype.h | ※5 ctype.h | ctype.h | ctype.h | ○ | ○ | ctype.h |
| isalpha | ctype.h | ctype.h | ※5 ctype.h | ctype.h | ctype.h | ○ | ○ | ctype.h |
| isascii | ctype.h | ctype.h | ※5 ctype.h | ctype.h | ctype.h | ○ | ○ | ctype.h |
| iscntrl | ctype.h | ctype.h | ※5 ctype.h | ctype.h | ctype.h | ○ | ○ | ctype.h |
| isdigit | ctype.h | ctype.h | ※5 ctype.h | ctype.h | ctype.h | ○ | ○ | ctype.h |
| isgraph | | ctype.h | ※5 ctype.h | | ctype.h | | ○ | ctype.h |
| islower | ctype.h | ctype.h | ※5 ctype.h | ctype.h | ctype.h | ○ | ○ | ctype.h |
| isprint | ctype.h | ctype.h | ※5 ctype.h | ctype.h | ctype.h | ○ | ○ | ctype.h |
| ispunct | ctype.h | ctype.h | ※5 ctype.h | ctype.h | ctype.h | ○ | ○ | ctype.h |
| isspace | ctype.h | ctype.h | ※5 ctype.h | ctype.h | ctype.h | ○ | ○ | ctype.h |

| 関数名 | 4.2BSD | MS-C | Lattice | MWC | HI-TECH | DeSmet | RUN/C | LSI C |
|---|---|---|---|---|---|---|---|---|
| isupper | ctype.h | ctype.h | ※5 ctype.h | ctype.h | ctype.h | ○ | ○ | ctype.h |
| isxdigit | ctype.h | ctype.h | ※5 ctype.h | | ctype.h | | ○ | ctype.h |
| toascii | ctype.h | ctype.h | ※5 ctype.h | ctype.h | ctype.h | | | |
| tolower | ctype.h | ctype.h | ※5 ctype.h | ctype.h | ctype.h | ○ | ○ | ctype.h |
| toupper | ctype.h | ctype.h | ※5 ctype.h | ctype.h | ctype.h | ○ | ○ | ctype.h |
| _tolower | | ctype.h | | ctype.h | | | | |
| _toupper | | ctype.h | | ctype.h | | | | |
| strcat | string.h | string.h | string.h | ○ | ○ | ○ | ○ | string.h |
| strcmp | string.h | string.h | string.h | ○ | ○ | ○ | ○ | string.h |
| strcmpi | | string.h | string.h | | | | | |
| strcpy | string.h | string.h | string.h | ○ | ○ | ○ | ○ | string.h |
| strdup | | string.h | string.h | | | | | |
| strins | | | string.h | | | | | |
| strlen | string.h | string.h | string.h | ○ | ○ | ○ | ○ | string.h |
| strncat | string.h | string.h | string.h | ○ | ○ | ○ | | string.h |
| strncmp | string.h | string.h | string.h | ○ | ○ | ○ | | string.h |
| strncpy | string.h | string.h | string.h | ○ | ○ | ○ | | string.h |
| strnset | | string.h | string.h | | | | | |
| strset | | string.h | string.h | | | | | |
| index | string.h | | | ○ | ○ | ○ | | |
| rindex | string.h | | | ○ | ○ | ○ | | |
| strchr | | string.h | string.h | | ○ | | | string.h |
| strcspn | | string.h | string.h | | | | | string.h |
| strpbrk | | string.h | string.h | | | | | string.h |
| strrchr | | string.h | string.h | | ○ | | | string.h |
| strspn | | string.h | string.h | | | | | string.h |
| strtok | | string.h | string.h | | | | | string.h |
| atof | math.h | math.h | math.h | math.h | math.h | ○ | ○ | stdlib.h |
| atoi | ○ | stdlib.h | stdlib.h | ○ | ○ | ○ | ○ | string.h |
| atol | ○ | stdlib.h | stdlib.h | ○ | ○ | ○ | ○ | string.h |
| ecvt | ○ | stdlib.h | math.h | ○ | | | | |
| fcvt | ○ | stdlib.h | stdlib.h | ○ | | | | |
| gcvt | ○ | stdlib.h | stdlib.h | ○ | | | | |
| itoa | | stdlib.h | ◇ | | | | | |
| ltoa | | stdlib.h | | | | | | |
| ultoa | | stdlib.h | | | | | | |
| strlwr | | string.h | string.h | | | | | |

※5 …… fctype.h でも定義されている（ctype.h ではマクロ定義／fctype.h では関数定義）

| 関数名 | 4.2BSD | MS-C | Lattice | MWC | HI-TECH | DeSmet | RUN/C | LSI C |
|---|---|---|---|---|---|---|---|---|
| strrev | | string.h | string.h | | | | | |
| strupr | | string.h | string.h | | | | | |
| **算術関数** | | | | | | | | |
| abs | ○ | stdlib.h | math.h | ○ | ○ | ○ | | stdlib.h |
| acos | math.h | math.h | math.h | math.h | math.h | ○ | ○ | |
| asin | math.h | math.h | math.h | math.h | math.h | ○ | ○ | |
| atan | math.h | math.h | math.h | math.h | math.h | ○ | ○ | |
| atan2 | math.h | math.h | math.h | math.h | math.h | | ○ | |
| cabs | math.h | math.h | | math.h | | | | |
| ceil | math.h | math.h | math.h | math.h | math.h | ○ | ○ | |
| cos | math.h | math.h | math.h | math.h | math.h | ○ | ○ | |
| cosh | math.h | math.h | math.h | math.h | math.h | | ○ | |
| exp | math.h | math.h | math.h | math.h | math.h | ○ | ○ | |
| fabs | math.h | math.h | math.h | math.h | math.h | ○ | ○ | |
| floor | math.h | math.h | math.h | math.h | math.h | ○ | ○ | |
| fmod | | math.h | math.h | | | | | |
| frexp | math.h | math.h | math.h | math.h | math.h | ○ | | |
| hypot | math.h | math.h | | math.h | | | | |
| iads | | | stdlib.h | | | | | |
| labs | | stdlib.h | stdlib.h | | | | | |
| ldexp | math.h | math.h | math.h | math.h | math.h | ○ | | |
| log | math.h | math.h | math.h | math.h | math.h | ○ | ○ | |
| log10 | math.h | math.h | math.h | math.h | math.h | ○ | ○ | |
| matherr | | math.h | math.h | | | | | |
| modf | math.h | math.h | math.h | math.h | math.h | ○ | | |
| pow | math.h | math.h | math.h | math.h | math.h | ○ | ○ | |
| rand | ○ | stdlib.h | math.h | ○ | ○ | ○ | ○ | stdlib.h |
| sin | math.h | math.h | math.h | math.h | math.h | ○ | ○ | |
| sinh | math.h | math.h | math.h | math.h | math.h | | ○ | |
| sqrt | math.h | math.h | math.h | math.h | math.h | ○ | ○ | |
| srand | ○ | stdlib.h | math.h | ○ | ○ | ○ | ○ | stdlib.h |
| tan | math.h | math.h | math.h | math.h | math.h | ○ | ○ | |
| tanh | math.h | math.h | math.h | math.h | math.h | | ○ | |
| **DOS インターフェイス関数** | | | | | | | | |
| bdos | | dos.h | ○ | | ▽ | ▽ | | stdlib.h |
| bdosx | | | ○ | | | | | |
| dosexterr | | dos.h | | | | | | |
| int86 | | dos.h | dos.h | | dos.h | ▽ | | |
| int86x | | dos.h | dos.h | ▽ | dos.h | | | |

| 関数名 | 4.2BSD | MS-C | Lattice | MWC | HI-TECH | DeSmet | RUN/C | LSI C |
|---|---|---|---|---|---|---|---|---|
| intdos | | dos.h | dos.h | | dos.h | ○ | | |
| intdosx | | dos.h | dos.h | | dos.h | | ○ | |
| segread | | dos.h | ○ | | dos.h | ◇ | | |
| inp | | conio.h | dos.h | ▽ | ○ | ▽ | ○ | stdrom.h |
| outp | | conio.h | dos.h | ▽ | ○ | ▽ | ○ | stdrom.h |
| FP_OFF | | dos.h | dos.h | ◇ | dos.h | | | |
| FP_SEG | | dos.h | dos.h | ◇ | dos.h | | | |
| cgets | | conio.h | ○ | | ○ | | | |
| cprintf | | conio.h | stdio.h | | | | | |
| cputs | | conio.h | ○ | | ○ | | | |
| cscanf | | conio.h | stdio.h | | | | | |
| getch | | conio.h | dos.h | | ○ | ▽ | ○ | |
| getche | | conio.h | dos.h | | ○ | | | |
| putch | | conio.h | dos.h | | ○ | ▽ | ○ | |
| ungetch | | conio.h | dos.h | | ○ | | ○ | stdio.h |
| kbhit | | conio.h | dos.h | | ○ | | ▽ | stdio.h |
| **その他の関数** | | | | | | | | |
| assert | assert.h | assert.h | assert.h | assert.h | assert.h | | | |
| getenv | ○ | stdlib.h | stdlib.h | ○ | ○ | | | |
| putenv | | stdlib.h | stdlib.h | | | | | |
| longjmp | setjmp.h | setjmp.h | setjmp.h | ○ | setjmp.h | ○ | | setjmp.h |
| setjmp | setjmp.h | setjmp.h | setjmp.h | ○ | setjmp.h | ○ | | setjmp.h |
| isatty | ○ | io.h | fcntl.h | | ○ | | | |
| perror | ○ | stdlib.h | stdio.h | | ○ | | | |
| poserr | | | dos.h | | | | | |
| swab | ○ | stdlib.h | | ○ | | | | |
| bserch | | search.h | | | | | | |
| qsort | ○ | search.h | stdlib.h | ○ | ○ | ○ | | stdlib.h |

表A-3　各処理系のインクルードファイル一覧表

## ■「stdio.h」ファイルの比較

インクルードファイルのなかで最も使用頻度が高い「stdio.h」ファイルの中身について，以下に比較表を示します．ここで，紹介するのはおもな定数の定義とFILE構造体の中身です．

| 処理系 | | 4.2BSD | MS-C | Lattice | MWC | HI-TECH | DeSmet | LSI C |
|---|---|---|---|---|---|---|---|---|
| 定数 | EOF | −1 | −1 | −1 | −1 | −1 | 0 | −1 |
| | NULL | 0 | 0 | 0 | (char *)0 | 0 | −1 | 0 |
| | BUFSIZ | 1024 | 512 | 512 | 1>>9 | 1024 | × | 1024 |
| | _NFILE | 20 | 20 | 20 | 30 | 15 | × | × |
| 標準入出力 | stdin | &_iob[0] | &_iob[0] | &_iob[0] | &_stdin | &_iob[0] | 0 | &_iob[0] |
| | stdout | &_iob[1] | &_iob[1] | &_iob[1] | &_stdout | &_iob[1] | 1 | &_iob[1] |
| | stderr | &_iob[2] | &_iob[2] | &_iob[2] | &_stderr | &_iob[2] | 2 | &_iob[2] |
| その他の定義 | | | stdaux &_iob[3] | stdaux &_iob[3] | CTRLZ 26 | | TRUE 1 | SYS_OPEN 10 |
| | | | stdprn &_iob[4] | stdprt &_iob[4] | | | FALSE 0 | WILDCARD −2 |
| | | | | | | | ERR −1 | |
| | | | | | | | stdprn 4 | |

<FILE構造体の宣言>

4.2BSD

```
extern  struct  _iobuf {
        int     _cnt;
        char    *_ptr;
        char    *_base;
        int     _bufsiz;
        short   _flag;
        char    _file;
} _iob[_NFILE];

#define FILE    struct _iobuf
```

MS-C

```
#define FILE    struct _iobuf

extern  FILE {
        char    *_ptr;
        int     _cnt;
        char    *_base;
        char    _flag;
        char    _file;
} _iob[_NFILE];
```

Lattice

```
struct  _iobuf {
        unsigned char  *_ptr;
        int     _rnct;
        int     _wcnt;
        unsigned char  *_base;
        int     _size;
        int     _flag;
        unsigned char  _file;
        unsigned char  _cbuff;
};

extern  struct _iobuf _iob[_NFILE]

#define FILE    _iob
```

MWC

```
typedef struct  FILE {
        unsigned char  *_cp,
                       *_dp,
                       *_bp;
        int     _cc;
        int     (*_gt)(),
                (*_pt)();
        int     _ff;
        char    _fd;
        int     _uc;
} FILE;
```

HI-TECH

```
extern  struct   _iobuf {
        char *   _ptr;
        int      _cnt;
        char *   _base;
        uchar    _flag;
        char     _file;
} _iob[_NFILE];

#define FILE     struct _iobuf
```

DeSmet

```
typedef int      FILE;
```

LSI C

```
typedef struct {
        char     mode;
        char     *ptr;
        int      rcount;
        int      wcount;
        char     *base;
        unsigned bufsiz;
        int      fd;
        char     smallbuf[1];
} FILE;
```

表A-4　stdio.h ファイルの比較表

## ■各コンパイラのオプション一覧

以下では，各C言語コンパイラの「プリプロセッサ制御」，「デバッグ」，「メモリモデルの指定」に関するオプションの一覧表を示します．

表の見方

1：プリプロセッサへの指示についてのオプション
2：デバッグをするときに便利なオプション
3：メモリモデルを指示するオプション

## 4.2BSD

| | オプション | 機能 |
|---|---|---|
| 1 | -E | プリプロセッサのみを実行し，標準出力に書き出す |
| | -C | プリプロセッサ処理時にコメントを取り除かない |
| | -D 識別子[＝文字列] | プリプロセッサに識別子の定義を指示する．文字列を省略した場合は1と定義される |
| | -U | 定義済みの識別子の定義を除去する |
| | -I パス名 | インクルードファイルの存在するディレクトリを指定する |
| 2 | -g | デバッガ用のシンボル情報をオブジェクトファイルに付加する |
| | -p | プログラムの実行時に関数が呼び出された回数と時間の情報を「mon.out」に出力する |
| | -S | アセンブラソースファイルを生成する．拡張子は「.s」 |

〈注意〉・デバッグに関するオプションはこのほかにもあるが，ここではおもなオプションのみを取り上げた．

## MS-C

| | オプション | 機能 |
|---|---|---|
| 1 | /C | プリプロセッサ処理時にコメントを保存する(/E,/P,/EPと合わせてのみ使用される) |
| | /D 識別子[＝文字列] | プリプロセッサに識別子の定義を指示する．文字列を省略した場合は1と定義される |
| | /E | プリプロセッサのみを実行し，その結果を標準出力に出力する．このとき#line命令を各インクルードファイルの最初と最後に挿入する |
| | /EP | プリプロセッサのみ実行し，#line命令を挿入せずに標準出力にその結果を出力する |
| | /Ｉパス名 | インクルードファイルの存在するディレクトリを指定する |
| | /P | プリプロセッサのみ実行し，#line命令を挿入せずに「.Ｉ」の拡張子を付けてファイルに出力する |
| | /U | あらかじめ定義されている以下の識別子の定義をすべて除去する<br>・MSDOS<br>・M_186<br>・M_186xM …xはメモリモデルを示すS,M,Lのうちのいずれか1つ<br>・SS_NE_DS…オプションでスタックとデータセグメントを分離することを指定したときにのみ定義される |
| | /U 識別子 | あらかじめ定義されている上記の識別子の定義を除去する |
| 2 | /Fa [ファイル名] | アセンブラソースファイルを生成する．拡張子は「.asm」 |
| | /Fc [ファイル名] | ソースとアセンブラの結合リストを生成する．拡張子は「.cod」 |
| | /Wn | コンパイラの警告メッセージの出力レベル（n=0〜3）をセットする |
| | /w | コンパイラの警告メッセージの出力レベルを0にする |
| | /Zd | 行番号の情報をオブジェクトファイルに加える |
| | /Zg | ソースファイルで定義されている関数の宣言を標準出力に出力する |
| | /Zs | 文法チェックだけを行う |
| 3 | /A 文字 | メモリモデルの指定（文字はS, M, Lのうちいずれか1つ） |

〈注意〉　・この表は「MSC.EXE」のオプションをもとに作成した．

## HI-TECH

| | オプション | 機能 |
|---|---|---|
| 1 | -Ｉ名前 | インクルードファイルの存在するユーザー・エリアおよびドライブを指定する |
| | -D 識別子[＝文字列] | プリプロセッサに識別子の定義を指示する．文字列を省略した場合は1と定義される |
| | -U 識別子 | 定義済みの識別子の定義を除去する |
| 2 | -S | 入力されたすべてのC言語ファイルをコンパイルし，アセンブラ出力を残す |
| | -V | コンパイラの各ステップの実行状況がCRT上に表示される |
| | -F | デバッガを使うために，リンカにシンボルファイルの出力を要求する |
| 3 | -B | ラージモデルを指定する |

## Lattice

| | オプション | 機能 |
|---|---|---|
| 1 | -d 識別子[＝文字列] | プリプロセッサに識別子の定義を指示する．文字列を省略した場合は1と定義される |
| | -u[識別子] | あらかじめ定義されている以下の識別子の定義を除去する．識別子を指定しないときは，すべての定義が除去される<br>・MSDOS<br>・i8086<br>・i8086x … x はメモリモデルを示す S,P,D,L のうちのいずれか1つ<br>・SPTR …… S,P モデルのとき定義される<br>・LPTR …… D,L モデルのとき定義される<br>・SFLG …… LC1 で -s オプションを指定すると定義される<br>・DEBUG … -d オプションを指定すると定義される |
| | -i パス名 | インクルードファイルの存在するディレクトリを指定する |
| 2 | -d[数] | デバッグのための情報を出力する．[数]には以下の3つが指定できる．値を省略した場合は0を指定したとみなされる<br>・0……ソースファイルの行番号情報を出力する<br>・1……行番号とローカル・シンボルを出力する<br>・2……行番号とローカル・シンボルとシンボルの型の情報を出力する |
| | -p | プリプロセッサのみを実行し，その結果を「.p」の拡張子を付けてファイルに出力する |
| 3 | -mM | メモリモデルを指定する（M は S,P,D,L または 0,1,2,3 のうちいずれか1つ） |

〈注意〉 ・この表は「LC1.EXE」のオプションをもとに作成した．

## MWC

| | オプション | 機能 |
|---|---|---|
| 1 | -d 識別子[＝文字列] | プリプロセッサに識別子の定義を指示する．文字列を省略した場合は1と定義される |
| | -Ｉ パス名 | インクルードファイルの存在するディレクトリを指定する |
| | -U 識別子 | 定義済みの識別子の定義を除去する |
| 2 | -c | リンクを行わずに，コンパイルないしはアセンブルのみを行う |
| | -k | 中間ファイルを消さずに残す |
| | -v | コンパイルの進行状況を出力する |
| | -vasm | アセンブラソースファイルをオブジェクトファイルの代わりに出力する |
| | -vdebug | debug のための情報を出力する |
| | -vquiet | strict check による警告をいっさい行わない |
| | -vsbook | K&R にない拡張をすると警告する |
| | -vsmemb | 構造体と共用体のメンバーに関してのチェックを行う |
| 3 | -vlarge | ラージモデルを指定する |
| | -vsmall | スモールモデルを指定する |

## DeSmet

| | オプション | 機能 |
|---|---|---|
| 1 | -I パス名 | インクルードファイルの存在するディレクトリを指定する |
| | -n 識別子[＝文字列] | プリプロセッサに識別子の定義を指示する．文字列を省略した場合は1と定義される |
| 2 | -A | アセラブラソースファイルを生成する |
| | -C | デバッガのための情報を生成す |

## LSI C

| | オプション | 機能 |
|---|---|---|
| 1 | -d 識別子[＝文字列] | プリプロセッサに識別子の定義を指示する |
| 2 | -o ドライブ名 | 中間言語ファイルを指定したドライブに作る |
| | -s | ソースプログラム中にエラーが検出されたときにも，「$$$.sub」を消さずにサブミットファイルの実行を続行させる |

〈注意〉　・この表は「CF.COM」のオプションをもとに作成した．

**表A－5　各コンパイラのオプション一覧表**

## ■ 8086系CPUの概説

C言語のプログラムを組む上でCPUの知識が必要となる場合があります．そこで，最近多くの16ビットパソコンで使われている8086系CPUについて簡単に解説しておきます．

### ― レジスタ構成 ―

8086CPUのレジスタは基本的に16ビット長で構成され，以下の図に示すように特定のレジスタが専用レジスタとして使われる場合がほとんどです．つまりレジスタの用途が明確に定まっているわけです．

8086系CPUの特徴は，プログラムやデータを64Kバイトごとの**セグメント**と呼ばれる単位に区切って管理することです．また，セグメント内でのメモリのアドレスは**オフセット**と呼ばれます．このセグメントの管理に使われるのが，セグメント・レジスタと呼ばれる4つのレジスタです．たとえば，現在プログラムが実行されているアドレスはCSレジスタの指し示すセグメントのIPレジスタが示すアドレス(オフセット)として表されます．

### ― アドレスの指定 ―

8086系CPUが直接扱えるメモリエリアの大きさは1Mバイトであり，物理的に20ビットのアドレス指定によって，すべてのメモリが表せます．ここで特定のメモリの指定は前述の「セグメント」と「オフセット」(ともに16ビットずつ)を使って行います．つまり32ビットのアドレスの指定で20ビットのアドレスを表すことになるので，以下のようなアドレスの指定方法が取られています．

16ビット（8ビット｜8ビット）

| | | | | | |
|---|---|---|---|---|---|
| AX: | AH | AL | アキュムレータ・レジスタ | 上位／下位8ビットずつ別々にアクセスすることが可能 | 汎用レジスタ |
| BX: | BH | BL | ベース・レジスタ | | |
| CX: | CH | CL | カウンタ・レジスタ | | |
| DX: | DH | DL | データ・レジスタ | | |
| | SP | | スタック・ポインタ | スタックの制御に使われる | |
| | BP | | ベース・ポインタ | | |
| | SI | | ソース・インデックス | データの転送などに使われる | |
| | DI | | ディスティネーション・インデックス | | |
| | IP | | インストラクション・ポインタ | ……現在プログラムが実行されているアドレス(オフセット) | |
| | $FLAGS_H$ | $FLAGS_L$ | フラグ・レジスタ | ……プログラムの分岐の制御など(以下の図を参照) | |
| | CS | | コード・セグメント・レジスタ | ……現在プログラムが実行されているセグメント | セグメント・レジスタ |
| | DS | | データ・セグメント・レジスタ | ……プログラムで扱うデータのセグメント | |
| | SS | | スタック・セグメント・レジスタ | ……プログラムで扱うスタックのセグメント | |
| | ES | | エクストラ・セグメント・レジスタ | ……プログラマが自由に使えるセグメント | |

〈フラグレジスタの各ビットの内容〉

| HIGH 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | LOW 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ＼ | ＼ | ＼ | ＼ | OF | DF | IF | TF | SF | ZF | ＼ | AF | ＼ | PF | ＼ | CF |

- OF：オーバーフロー・フラグ
- DF：ディレクション・フラグ
- IF：インタラプト・フラグ
- TF：トラップ・フラグ
- SF：サイン・フラグ
- ZF：ゼロ・フラグ
- AF：補助キャリー・フラグ
- PF：パリティ・フラグ
- CF：キャリー・フラグ

**図A－1　8086CPUのレジスタ構成**

**図A－2　セグメント／オフセットと物理アドレスの関係**

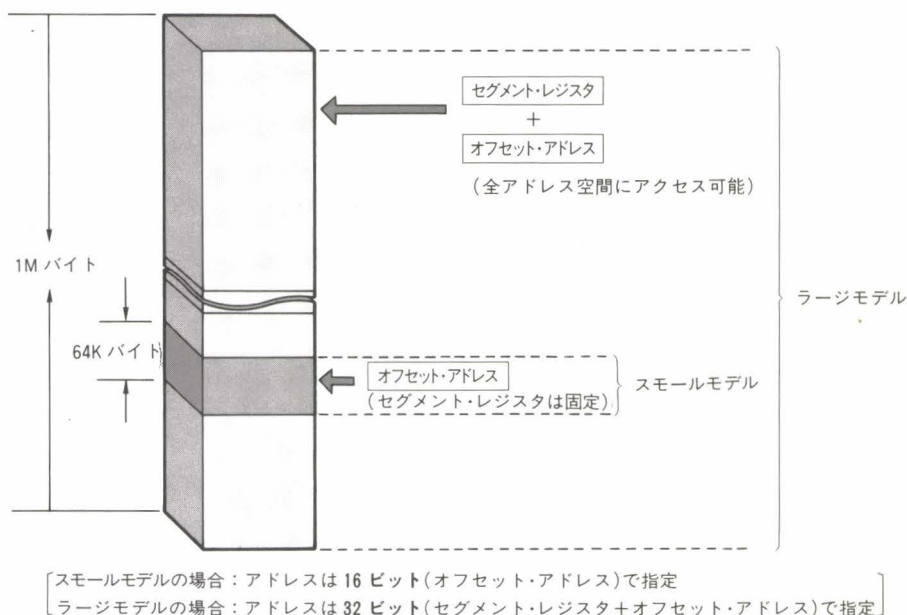
**― スモールモデルとラージモデル ―**

C言語でプログラムを作成し，コンパイルする場合は次に示すメモリモデルを指定しなければなりません（指定方法は「各コンパイラのオプション一覧」を参照）．

・**スモールモデル** ……64Kバイト(同一セグメント)内ですべてのプログラムとデータを扱う
・**ラージモデル** ………64Kバイト(同一セグメント)を越えるプログラムとデータを扱う
・**ミディアムモデル**† … 上記の2つの中間的な扱いをする

通常作成する規模のプログラムではスモールモデルを指定します．これに対して，エディタやデータベースなど64Kバイトを越える非常に大きなプログラムや多量のデータを扱う場合はラージモデルでコンパイルしなければなりません．なお小さなプログラムでもラージモデルでコンパイルすることは可能ですが，その場合は実行ファイルのサイズが大きくなります．また49ページの**リスト2-1**で示したような全アドレス空間をアクセスするプログラムはラージモデルでコンパイルします．

以下にラージモデルとスモールモデルの概念図を示しておきます．

[スモールモデルの場合：アドレスは**16ビット**(オフセット・アドレス)で指定
ラージモデルの場合：アドレスは**32ビット**(セグメント・レジスタ＋オフセット・アドレス)で指定]

**図A-3　スモールモデルとラージモデル**

† ミディアムモデルはサポートしていない処理系もある．

# 索　引

**【参考文献】**


・「プログラミング言語C」　B.W.Kernighan, D.M.Ritchie 共著　石田晴久訳　　共立出版
・「C言語入門」　L.Hancock, M.Krieger 共著　アスキー出版局監訳　　アスキー出版局
・「C言語プログラミング入門　UNIX 版」　西田，五月女共著　　啓学出版
・「UNIX System V　ユーザ・リファレンス・マニュアル」　日本ユニソフト㈱訳　　共立出版
・「Microsoft C　ユーザーズガイド，ランゲージリファレンス」　マイクロソフト㈱
・「Microsoft C　ランタイム　ライブラリ　リファレンス」　マイクロソフト㈱
・「アルゴリズム＋データ構造＝プログラム」　N.Wirth 著　片山卓也訳　日本コンピュータ協会


**プログラム協力（第10章「xref」／「less」）**

池田　けんしろう

**実習C言語**


アスキー・ラーニングシステム②実習コース

1986年11月11日　初版発行
1988年4月11日　第1版第6刷発行
定価1,800円

著　者　三田典玄（みたのりひろ）
発行者　塚本慶一郎
発行所　株式会社アスキー
　〒107 東京都港区南青山6-11-1スリーエフ南青山ビル
振　替　東京4-161144
TEL (03)486-7111（大代表）
情報 TEL (03)498-0299（ダイヤルイン）
出版営業部 TEL (03)486-1977（ダイヤルイン）





編集担当　佐藤英一
表紙担当　郷　啓子
CTS　福田工芸株式会社
印刷　株式会社加藤文明社印刷所

ISBN4-87148-227-8 C3055 ¥1800E